# PROGRAMMING A DIALOGUE TEACHING SITUATION

BY

JAMES L. STANSFIELD.

PH.D. THESIS. UNIVERSITY OF EDINBURGH.

1974

## ABSTRACT

This thesis resulted from a critical examination of how techniques from artificial intelligence might help towards writing an intelligent teaching program. The two problems it covers are how to represent knowledge, and how to organise processes.

Part I presents a theoretical approach to knowledge representation. Chapters 1 to 3 discuss the theory, which is based upon a principle called the distribution principle. This states that knowledge should have procedural content and should be distributed throughout an intelligent system. A method of implementing this principle is given. It requires that procedures be grouped into clusters and a hierarchical organisation for these is discussed. Chapters 4 to 6 each describe a program which illustrates some aspect of the theory in a practical situation. The programs include such topics as using dialogue; models of one's partner in a conversation; answering "why" questions; and redundancy and inconsistencies in a knowledge base.

The procedural approach to representing knowledge taken in Part I creates problems concerning the organisation of interacting processes. Part II, then, reviews control structures, which are one means designed to overcome these problems. A powerful generalisation of control structures, called P-74, is presented. P-74 helps to unify existing techniques by introducing a new concept of layers of control structure.

The conclusion of Part II is that control structures are restrictive, and a freer approach is needed. A possible primitive for this approach is given and is related to the message-sending primitive used in ACTOR systems (Hewitt, 1973). Since the theory of knowledge presented in Part I had much in common with ACTORS, the two parts of the thesis have similar conclusions.

ACKNOWLEDGEMENTS

I should like to express my thanks and indebtedness to the many colleagues and friends who have helped me in a multitude of ways throughout this project. In particular, I should like to thank

- my supervisor, Jim Howe
- Gordon Plotkin and Carl Hewitt for reading the draft copy
- Sylvia Weir for encouragement and perspective
- Mike Gordon, Nat Goodman, John Knapman, Tim Radford and Marc Eisenstadt for discussion on P-74
- Stephen Isard and Tony Davey, for valuable comments about representing language
- Harry Barrow, George Kiss and Richard Young, for discussing processes and representation
- Cathie Sutherland and Alison Morris, for typing the draft copy
- Margo Jayes for so expertly typing the final version
- IBM United Kingdom Limited and the Social Science Research Council, for financial support
- and all those who by lectures, papers or simply being around created a feeling that Artificial Intelligence is exciting.

C O N T E N T S

PART 2 Control Structures

Prologue

This research is related to teaching and learning. Along with Minsky and Papert (1972), we assert that the modern computer and its associated theory of computation provides the only framework we have rich enough to express useful detailed ideas about intellectual processes. About a science of intelligence, Minsky and Papert say

> "But just as astronomy succeeded astrology, following Kepler's discovery of planetary regularities, the discoveries of these many principles in empirical explorations of intellectual processes in machines should lead to a science, eventually."

As this science emerges, we may be able to produce intelligent programs which understand about learning and teaching. At present, except perhaps in some special cases, teaching programs must lack the understanding needed to instruct, advise, guide or accomplish the many other facets of teaching.

Our research began in 1970 as a reaction to computer assisted instruction. Programs in this field hardly used the techniques that were appearing in artificial intelligence and did little to merit the term "teaching program". Instead, they seemed to be fact dispensers, automatic page turners, or statistics gatherers. Trying to remedy this, we set out with the ambitious aim of producing a dialogue teaching program which really knew what it was doing and which was a good model of teaching, learning, language use, and knowledge representation. It soon became clear that this aim was beyond the state of the art and that we should make a deeper investigation into some of the problems that arose. It is this investigation which we report here.

It was evident from our early work (Stansfield, 1971) that our main problem was how to represent procedural knowledge. This includes both knowledge of processes themselves and procedural knowledge about static facts. We had thought that by organizing and interconnecting facts into a sophisticated relational data-base we could represent verbal and informally structured facts in a useful and suggestive way. In the event, such a data-base was insufficient for a teaching program which aims to rise above the paradigm of pumping facts from one bucket to another. The work showed that we need

to organise knowledge differently. The emphasis must shift from nouns like "geography" to the more active viewpoint of how to be a geographer, a teacher or a language user.

Part I of this thesis is an investigation into representations of knowledge, its uses, and the difficulties in communicating it and explaining it in dialogues between a human and a computer.

It falls naturally into two sections. Chapters 1 to 3, forming the first section, give a theoretical discussion of the problem. They explain why our original representation failed. They discuss the respective advantages and disadvantages of procedural and declarative representations, and they go some way towards resolving the differences between these two viewpoints by considering knowledge to be represented as clusters of processes. Each cluster corresponds to a theme or concept. The processes in a cluster correspond to uses of the knowledge about that theme. We define a process and consider the implications of this definition upon communication between processes and grouping of processes. We argue that processes should be run in parallel and show how the operation of grouping parallel processes into clusters leads to a hierarchical organisation which avoids many problems that lead to combinatorial explosions of computation.

We introduce the principle of distribution of knowledge as the unifying principle underlying our theoretical discussion and show how clusters of processes fit in neatly with the principle.

Chapters 4 to 6 each describe programs which illustrate some part of the theoretical discussion. The programs are GEOG-LINE, CARTOGRAPHER, and a program which answers "why" questions. We can tell GEOG-LINE facts about its microworld and it will respond with a dialogue in order to satisfy its curiosity or resolve its internal contradictions. GEOG-LINE illustrates some principles about representing knowledge of dialogue, in particular that utterances must always be thought of in a context of goals, intentions and desired effects.

CARTOGRAPHER has a larger microworld than GEOG-LINE. It answers questions by enlisting the aid of its questioner for solving subproblems. This is the basis for its dialogue. It has a simple procedural model of its user's

knowledge so that is asks him reasonable questions. We use it to discover more about the kinds of model that are needed. CARTOGRAPHER also shows that consistency and non-redundancy are often detrimental properties for a knowledge base.

Our third program gives explanations in simple english for "why" questions represented as statements in a programming language. It illustrates the ideas of clusters of functions and distribution of knowledge. We also use it to illustrate the relationship between ACTORS (Hewitt, 1973) and clusters of functions.

Part 1I summarises and relates many different control structure facilities that are availabe in high-level programming languages. It includes back-tracking, co-routining, and the ENVEVAL primitive of Bobrow and Wegbreit (1973). We take these ideas and push them to their logical conclusion to produce a control regime which we call P-74. P-74 introduces the idea of layers of control structure which has not been referred to before although some two-layered languages do exist.

The facilities provided by other control regimes are combined into a unified scheme in P-74. We thought that this would result in a much more expressive programming language but the actual outcome was surprising. The higher layers of P-74 are cumbersome to use in much the same way as higher than second order logics are. Since P-74 is a logical completion of other control structures we argue that they themselves must be placed on a different basis. We argue that any uniform control structure is just a pattern of message passing. If this pattern is uniform in any particular set of examples then a regime can be extracted and can perhaps be more efficiently implemented. In general, however, greater freedom is needed as is provided in ACTOR systems.

# Part I Representing and Using Knowledge

## Chapter 1 Procedural Representations

### 1.1 Introduction

One of the major decisions that must be made in any practical Artificial Intelligence project is how knowledge is to be represented in the computer. This is especially true of programs which teach since the subject matter in this case is directly concerned with knowledge itself; how to express it, explain it, organise it, and model it in other people. Because of this, a major part of this research was concerned with the problem and we devote the first three chapters exclusively to the theoretical observations we made about knowledge representation. In later chapters we discuss programs which led to this theory and which embody parts of the theory in particular domains taken from our problem area of dialogue teaching situations.

Our presentation is theoretical not because there is any abstruse formalism present or because of masses of equations and difficult and rarified concepts. Instead, we use plain English augmented by computing concepts, and illustrate wherever possible by examples taken from real situations involving knowledge, rather than from pure artificial intelligence. Three illustrative programs are given in Chapters 4 to 6. We also make several definitions and state some underlying principles which we relate to other theories of knowledge currently being aired. These theories are : relational data bases, PLANNER procedural embedding, FRAME systems, and ACTOR systems.

One obvious way to represent a store of knowledge and one which has been tried many times, is to use a declarative relational retrieval net. We tried this approach ourselves and found many pitfalls which we discuss in this chapter. The chapter falls into two parts.

Firstly, we state four criticisms of relational data-bases and expand on each of them. In the main, the problems with such data-stores are that certain information, particularly procedural information, is not easily represented and that the information which is stored misleads the designer into feeling it is deeper than it really is because connotations which spring to his mind unthinkingly are based on the suggestiveness of English words used as data and not on knowledge in the store.

The second part of the chapter explains an alternative approach which answers the criticisms made about relational nets. It is based on the idea of gathering procedures which represent skills into bundles which have knowledge of how to use the skills and assimilate new ones. Perhaps the main thesis of our research is that knowledge of all kinds should be distributed throughout an intelligent system. We show how "bundles of skills" are a great help towards this. In the summary to this first chapter we illustrate this distribution principle by referring back to examples earlier in the chapter.

Chapters 2 and 3 develop the theme of representing knowledge using the principles put forward in Chapter 1. Chapter 2 investigates descriptions and the distinction between descriptions which have declarative content and those which have procedural content. In contrast to Chapter 1 we point out some advantages which relational structures have over procedures when used as descriptions. The principles of Chapter 1 are then applied to suggest a system which combines the best of both worlds and which we entitle active descriptions.

Since the idea of a process is fundamental to both Chapters 1 and 2 we devote Chapter 3 to a discussion of processes, how they can be represented and how they can communicate. An important section of this chapter discusses intensionality of data structures. A concept is intensionally defined if it is defined by its behaviour rather than by our behaviour upon it, i.e. by what it can do rather than what we can do to it. Intensionality is precisely the property we require of bundles of skills in Chapter 1.

## 1.2 Problems with Non-Procedural Representations

Until PLANNER (Hewitt, 1969, 1972), most data-bases were declarative relational data-structures built from words and links, and operated on by a set of uniform deduction and retrieval procedures, although there were some notable exceptions for instance SIR (Raphael, 1968). Examples of uniform data-bases are Quillian's Semantic Memory (Quillian, 1968), Simmon's Semantic Nets (Simmons, 1973), Carbonell's Geography Dialogue System (Carbonell, 1970), and our own Geography Data Base (Stansfield, 1971).

There are several problems inherent in such data-bases. They are :-

(1) Relations and objects are taken to be primitives.

(2) It is difficult to incorporate inference making.

(3) It is difficult for the data to guide the operating procedures.

(4) The facts by themselves do not do anything.

We will deal with these in turn.

### 1.2.1 Problem 1 Relations and Objects are taken to be primitives

The symptom of this problem is that complex relations such as that between an industry and a place are reduced to a simple relation such as "industry of". "Industry of" only has meaning to the system either by way of external qualifications such as "A is an important industry of B", or by its occurences in the data-base. But much of the meaning of "industry" should be in its internal structure. "A is an industry of B" is shorthand for one of many possible relationships with different structures. It is a first approximation.

Consider the meaning of the object "fishing". If we know that an industry of Newfoundland is fishing and if we ask "where are fish caught" a mistaken answer could be "in Newfoundland". The rule that

could produce this error is "If a place P has industry I and the industry produces product R then R is produced at P". However, fish are caught from and produced in the sea. By treating the relation syntactically, and ignoring the processes involved in fishing, we forget that fishermen leave Newfoundland in boats to catch fish which they then bring back to Newfoundland. The answer to the question is really found by examining what happens when fish are caught. Of course any system will make mistakes if it is given simplistic rules. However, relational data-bases encourage such rules since the mere presence of words like "industry", "boats", "catch", raise connotations effortlessly in the mind of the designer although the data-base itself has no capabilities for grasping these connotations. If we gave a dictionary to a Martian he would not be able to decipher the meanings of its contents. Anyone who reads a dictionary or examines a relational data-base brings to bear an enormous amount of knowledge which allows him to make use of it. If a data-base were capable of answering our question about fish, it would need to understand and manipulate procedures for, or descriptions of, such processes as "leave", "catch", "transport" and "produce".

Although relations can be used as primitives at times, just as the words used in a dictionary definition are temporarily regarded as primitives, to obtain good answers to questions, some of the relations involved in the questions should be analysed. If all relations are taken as primitives, then answers tend to be superficial since the real meaning of the relations and the key to the way they interact with other relations is inherent in their own behaviour.

This brings us to a key distinction between two viewpoints that can be taken when describing the behaviour of a system of inter-related objects. One viewpoint considers the objects as primitive and tries to describe them from outside, treating them merely as symbols in certain structures. We refer to this as the external viewpoint. The other viewpoint assigns structure to the objects, and their behaviour and relationships to other objects are expressible in terms of this structure. We call this an internal viewpoint. Here are some of the examples of the distinction.

Firstly, consider describing how chemicals react using rules which involve neither knowledge of valencies nor an electronic shell model of elements. The rules fit for many cases of a general nature but have anomalies. To incorporate the anomalies, exceptions to the rules are needed and there is an enormous increase in complexity of rules for diminishing returns in terms of the number of reactions described. On the other hand, a good model, delving into the structure of the elements involved, for example, a valency model, has far greater clarity and explanatory power and also gives insight into why the rules are as they are. A model is a feature of semantic explanations. Of course, an incorrect model may also produce extremely complicated rules with vast quantities of ad-hoc-ery. The epicycle theory of the Solar system illustrates this.

Secondly, consider the rules of transitivity for the subset relationship. Given any set of sets we can specify some subset relations between the sets and can use the transitivity rules to deduce further subset relations. We argue that there are other and perhaps more flexible ways in which people make deductions that appear to be using these rules. Our arguments use the idea of a "description" of a concept.

First, suppose that the elements which make up our sets are structured entities with properties like "red", "green", "large", "having wheels", etc. We can describe the sets in terms of the properties of their elements. Suppose, for simplicity's sake, that all our sets have simple property descriptions which are conjunctions of properties or their negations, e.g. the set of all furry non-red things. Given such a description of all the sets and elements we are in a position to answer questions such as "is this a that?", or "are all a's b's?", without recourse to transitivity rules. Thus, "is a swallow an animal?" can be answered by taking an example ideal swallow or description of one and simply applying an animal testing function to it. We don't need to follow a chain swallow - bird - animal, or any more refined version the taxonomist can supply for us. If we did, then the more refined the system the longer the chain and the more work there would be. Moreover, questions can be answered more fruitfully than by yes or no. For example, "are all x's y's?" can be answered by "yes, provided they are not furry!" if all properties on the description of the set of x's are also properties

of y's and y's description forbids furry things whereas x's description does not.

A system based on descriptions would certainly make deductions that agree with the set inclusion rules. However, it would also explain why the rules worked by giving a model for them. The set-theoretic rules describe the relations between concepts from outside whereas our alternative gives internal rules describing subset relations by examining the structure of the concepts they relate.

We can sum up what we have learnt about the two viewpoints applied to these examples. Rules which take an external viewpoint either involve an inappropriate model which cannot adapt to the idiosyncrasies of parts of the system or else they use a model which can adapt only because it is sufficiently general to apply to any system. Internal sets of rules on the other hand use an appropriate model since local variations and context sensitivity are easily represented. Such rules are often explanatory.

Consider a model of some aspect of society. We can certainly specify many external facts such as relationships between people, their properties or characteristics, and their membership of clubs, factories and other institutions, but we nevertheless require a working model or behavioural description of each person before we can bring the relationships to life. It may be possible to follow some external rule like a timetable to see what any group of people are doing in some cases, but in others this will depend on the interactions between the whims and moods of the members of the group. In this situation, the structure of a complex system's behaviour can be described best as the result of interactions between other systems which are less complex and whose behaviour is described by internal procedural models.

The advance brought about by the advent of computers has been the expansion of our ability to frame such internal structural models. In some cases the question whether a model is good or not can be answered by looking inside the system being described. Unfortunately, in other

cases, notably any intelligent systems, this is not possible and models must be constructed by detailed investigation of behaviour and assessed on the grounds of parsimony, agreement with observation and elegance.

1.2.2 Problem 2 It is difficult to incorporate inference making

The reason that it is difficult to incorporate inference making in relational net data-bases is that the inferences which need to be made about any domain are very dependent on the particular domain. Problem 3 which concerns the problems of communication between domain data and retrieval procedures means that the data cannot guide the deduction procedures sufficiently for its own peculiarities. Relational data-bases, because of this restriction, commonly allow very generally applicable deduction rules and this makes most inferences difficult to express. The restricted inference rules that have been the basis for the deductive power of many systems are the subconcept, superconcept rules. Certain concepts were considered to be more general than others in a vaguely defined way and then transitivity of "superconcept" was used to find examples satisfying certain simple constraints. These rules are not useful for expressing inferences such as "You can tell if an object is an island by seeing if it is surrounded by water" which are best expressed by procedures, for example to test for "surrounded by".

Furthermore, there is a dimension which has inference at one extremity and more general computation at the other. A rule for finding an example of an island may be

Step 1) Move to the west until you come to a coast.
Step 2) Move round the coast looking out to sea for a small object. Go to Step 2.

This complements the inference rule given above but is very little like one itself although it is clearly a procedure.

There are also cases where apparently simple declarative facts are best viewed procedurally, as the result of inferences in context. Consider the property "big". Now, whether an object is big or not depends very

much on context. A flea might be a big flea unlike the perhaps small dog it inhabits. However, we could prove that it is difficult to find the flea because it is small whereas it is easy to find the dog as it is large. Whether an object is large or not depends upon the context of a particular use of large. We could easily express this in a program to test for largeness by writing rules which examined the current context. It would be difficult to attach a property "large" to all large objects since the objects we would need to consider would be all uses of "large" referred to objects and not just all objects.

A similar example is the concept of a "spanner". An object may sometimes be usable as a spanner and sometimes not, depending on the situation. We could use a table fork as a spanner in some situations. However, we would lose information if we gave most things most labels because of obscure and extremely occasional uses. It would be possible for a procedure which could evaluate what was needed as a spanner in a particular situation to find a new relation for that situation. For example, a spanner is "an object that can be positioned to grip the nut that needs turning in the current context and which provides leverage." The two underlined conditions are best described by programs. Our point is that the type of descriptions we need require special local procedural knowledge to manipulate them. Moreover, they embody the reasons that this solution works as a spanner.

### 1.2.3 Problem 3 It is difficult for data to guide the operating procedures

The third problem can be introduced by extending an analogy due to Hayes and Winston, called the egg-timer analogy. Related arguments on the same problem have also been given in Anderson and Hayes (1971). Figure 1 shows an egg-timer representing a data-base.

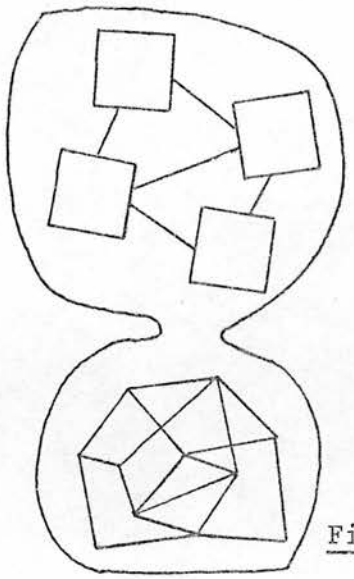


Figure 1

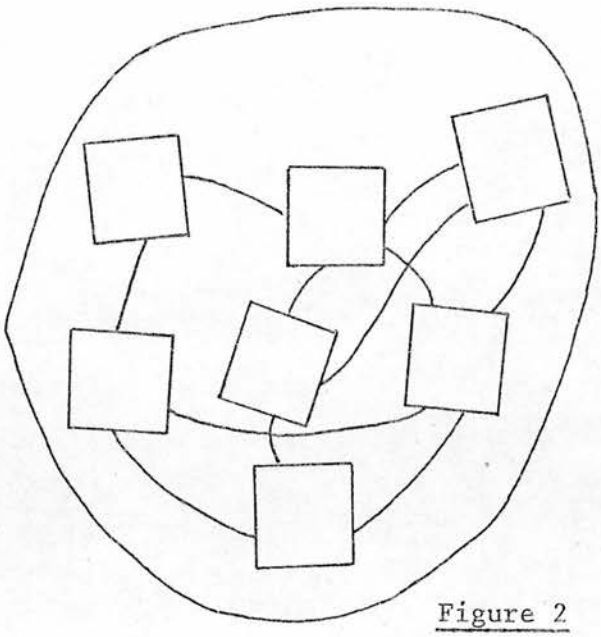


Figure 2

In the top half of figure 1, we have a set of boxes representing procedures. These represent the procedural knowledge the system has about its relations. They interconnect, since one procedure may call another. The procedures add new data, retrieve existing data, and make deductions. They are designed to work with data given in particular limited, conventional formats. Knowledge, in the form of data, is stored in the lower half of the egg-timer in a relational net of words and links between words.

The problem is that the procedural knowledge and the domain dependent knowledge are separated by the pinch in the egg-timer. The procedures can communicate with the data in very limited ways, for example by initiating searches or by asserting new links in the data. The conventions of communication are the same for any procedure communicating with any piece of data. They are uniform. Domain information has great difficulty in influencing the flow of the control of the procedures and hence the structure of the program's behaviour. The difficulty is that procedural knowledge properly associated with specific data objects cannot be written directly as programs. It must somehow be written into the structure of the data net in such a way as to influence the procedures in the upper half of the egg-timer. Moreover, these procedures cannot in any way be made to anticipate the interventions of the data since they are supposed to be general independent procedures.

We tried to solve this problem early in the research (Stansfield, 1971). While trying to represent geography in a relational net, we attempted to remove the communications barrier between procedures and domain dependent

knowledge by allowing extra formats in the net. To allow the data to guide the deduction done by the procedures we introduced special ways of representing rules of deduction as substructures of the relational data net. We also devised labelling methods to distinguish between instances of relational linkages that had differing interpretations to the search procedures. To cope with all these extra formats, the procedures in the upper half of the egg-timer had to be enhanced so that they could interpret structures using these formats.

We can now see this as a search for structures (the data net) in which procedural knowledge can be represented such that an interpreter (the procedural part) can execute that knowledge. But this makes the relationship between the procedures and the data the same as between an interpretive programming language and its program. So why not use a programming language anyway? Programming languages with control structures like LISP's or ALGOL's are not suitable. The reason is partly that subgoals such as "find an example of an island" can have several results. The first one may or may not be appropriate to the major goal though this can only be found out by trying it. To produce the second result, the program which found the first must continue from where it left off. PLANNER - 71 solved the problem by allowing a backtracking control structure. CONNIVER allowed generator functions which could be resumed. Our own control regime P - 74 was partly conceived by considering this problem.

This viewpoint allows us to squash the egg-timer into the shape of figure 2. We forget about the relational net. Instead of connections between data objects, we have a net of connections between procedures. Instead of each node being manipulated, it can be asked to do things. To produce such a net, we think of procedural knowledge we want about the objects in our domain. We think of what we want to do with each item. For example

what questions might be asked about it,
how to answer these questions,
what to do with new data of a certain type,
how to relate say, industries and places,
how to teach about temperature,

and so on. The interpretation of figure 2 is of a set of procedures, each being perhaps specific to a piece of real world knowledge. Each procedure can do certain actions or answer certain questions when called with the right arguments and may need to activate other procedures to do this. Thus the procedures are linked by procedure calls and this is represented in the diagram by links between the boxes.

Of course, with this model of a knowledge system the links need not be preset as in a relational net, but may also be implicit. For example, one box might need to call another to solve a problem and might then have the subproblem of finding which other box might be able to solve that problem. Another way we have implicit calls is if one procedure P, gives advice as well as a problem to another procedure Q, e.g. "Solve this, but if you get stuck ask Fred". Q's call to Fred was not explicitly set up at compile time.

An advantage of this approach is that a run-time control structure is generated as the procedures call each other. Whereas in figure 1, the data-base was searched by the uniform procedures in the top of the egg-timer, in the new situation the search space is generated by the data itself as each object calls others of its choice. It is possible that the behaviour of any box might depend on who called it, i.e. on the run-time environment.

### 1.2.4 Problem 4 The facts by themselves do not do anything

Relational data networks are first and foremost storage and retrieval devices. For the reasons given so far, relational data networks are inconvenient for representing procedural information so in themselves they cannot produce very interesting behaviour. Although they store facts, they do not understand them and although they can return facts on request, they cannot teach them or explain them. We will be returning to the problem of using knowledge of various types in later chapters, in particular, knowledge of simple maps, of language and dialogue, and of giving explanations. Here we just point out that a storage and retrieval system for facts is not a good primitive on which to build an intelligent system. It is important to consider how the system is intended to behave, how this can be procedurally specified, and then how facts can be

distributed through the procedures to accomplish particular local goals.

## 1.3 Clusters of functions

Now we have decided that intelligent systems should be structured more on the lines of figure 1) rather than of figure 2) let us examine in more detail the structure of each of the boxes in figure 2). We have already shown that they should have procedural content and that they need to remember factual information, pointers to other boxes, and their current state of control. They also need to contain procedures which allow them to respond to various requests relating to their subject matter.

Summing this up we can say

> We can gather together functions which relate to a common theme and use these clusters to represent the theme.

A theme is a cluster of procedures related to some subject matter. These clusters need information about how to utilize their skills and how to assimilate more. The knowledge could be from outside the cluster or inside it. To take an analogy from Sussman (1973), I could ask a plumber to do some job for me and give him advice (from outside of him) to help him. He would combine this advice with his own knowledge of how to use his skills (inside knowledge) and would try to complete the job.

We imagine a cluster of skills to be a process just as a plumber is. It is procedural and has a behaviour which is elicited by giving it requests. It also has an internal state which means its behaviour can be altered as time goes by. An intelligent system capable of many skills will be composed of such processes, themselves composed of many sub-processes and so on, down to primitive systems with very restricted knowledge. This hierarchical arrangement of processes (with each process being an agglomeration of other processes much as a human cell is a set of objects, each with its own behaviour, all contained within a membrane)

was one of the themes leading to P - 74, our control structure, which is based on similar hierarchical lines.

Clusters of skills have some similarities with the idea of a class in SIMULA (Dahl & Hoare, 1972). A class is basically a collection of data objects and ALGOL - 60 procedures encased in a module within which the items are locally named by identifiers. Any of the items may be retrieved by using the identifiers relative to the class. Any of the procedures may be called and after it has returned control it will still be possible to resume the procedure from its continuation point. In Part II, we explore the relationships between SIMULA and P - 74.

In more recent literature there are other ideas with similarities to clusters of skills. The ACTORS of Hewitt (1973) are objects which have the ability to receive messages in various forms and to act appropriately. The main difference between ACTORS and clusters of skills is that side-effects are reduced to a minimum in ACTORS. There is only one primitive which has side-effects (the cell) and that is only essential for true parallelism. We are not sure of the implications of this difference but feel intuitively that in the real world processes producing side-effects are the norm so the emphasis on lack of side-effects is misplaced.

Another recent relative is the FRAME idea of Minsky (Minsky, 1974) (Winograd, 1974). In the Winograd article, a frame is tentatively represented as a structure on which are hung procedures called imps. As information arrives at the frame the appropriate imps are activated and can both transmit results or requests to other frame or can modify their own frame. The parallels between clusters of functions and frames should be clear.

There is an interesting relation here with PLANNER procedures. For any goal pattern, a PLANNER program might have various methods of achieving the goal and these are stored in a data-base. We can consider the methods, which are antecedent and consequent theorems, to be the various skills of the cluster. However, since the methods are only indirectly linked by means of a data-base, it is difficult for the cluster to reorganise itself or build up advice on how to deploy its skills. Thus, we say that

PLANNER only has outside means of giving advice about how to coordinate the separate possibilities for achieving a goal. This takes the form of restrictions on the choice or order of procedures to be used and is given by the calling procedure at the time of a particular execution of any goal. It must not be confused with inside advice where the difference between our methods shows up.

Consider next the inside advice. In PLANNER there is one particular way in which procedures are coordinated for achieving goals. It consists of assuming separation of the skills and then trying them one after the other until one achieves the goal. This means it is difficult to locate advice for coordinating the procedures. In our approach, for any particular goal that is related to a concept we have a cluster of procedures which will bear upon that goal. Since these are all collected together we have an advantage over the situation in PLANNER. The cluster provides a very good attachment point for inside advice.

The motivation for the separation of related procedures in the design of PLANNER seems to have been that it would lead to a modular system which allowed programs to be easily incremented and adapted. Previous AI programming languages did not easily allow such alterations since function calls were always made explicitly. The idea of having several methods for the achievement of a goal called for a loosening of this explicit function call reference. It must not be overlooked that knowledge of how to put together skills effectively is also needed and that this knowledge is very dependent on the particular skills.

Although we take the position that new skills require assimilation, the benefits of modularity can be retained. They simply require that any cluster of skills should have particular skills relating to assimilation. The locality of this allows for modularity. More general knowledge about assimilation could be called upon by any cluster which needs it.

A new communications problem arises. In most programming languages when we call a function we give it a few simple arguments. We assume that the function is special purpose and that it knows what data to act on. Since

a cluster bar has a repertoire of skills, it needs more complex instructions on how to proceed and what it is required to do. To return to the example of the plumber. We could tell the plumber what we want him to do, where to do it, how fast, what resources we could spare, dangers in the house, our diagnosis of the fault with a specification of its symptoms, and so on. A tentative hypothesis we would like to put forward is that a type of programming language is needed in which we would express requests for subprograms rather like we do in English. This is as opposed to a language where communication between procedures is simple and the procedures describe what happens.

## 1.4 The Distribution Principle

An important principle has been exemplified throughout this chapter and we call it the distribution principle.

> The distribution principle states that knowledge of all kinds should be distributed throughout an intelligent system, by associating each individual item with the group of procedures which uses it.

A good method of representing knowledge should be compatible with this principle by providing many appropriate locations for knowledge. This does not preclude the use of directories for knowledge which needs to be generally available. Such directories have been proposed by Sloman (1972) for correlating processes with their purposes. The important consideration should always be the extent that the knowledge is used. Often we need to duplicate knowledge. For example, a city has many telephone directories. Most people have phone-books to record their important numbers and they also remember the very important ones. Distribution of knowledge in this way, according to its use, prevents exponential explosions as the amount of knowledge increases.

A further advantage of distributing knowledge is that processes which use knowledge in special ways can have special procedural representations for those ways. Consider knowledge about picking up objects. There can

be general procedures for picking up anything, and these may need to invoke special procedures for picking up furniture-shaped objects. An expert removal man is likely to have much special purpose knowledge about picking up objects.

There are four places in this chapter where the distribution principle has been mentioned implicitly.

Firstly, in Section 1.2.1 we considered internally modelled systems as opposed to externally modelled ones. Models of the objects represented in such systems were one of their most important features. We gave an example where knowledge about society was being represented and suggested that individual people within the society should be represented by models. Thus, knowledge about people was located, procedurally, within the individual people, and those social rules which were general were appropriately associated with the society as a whole.

Secondly, in Section 1.2.2, our definition of "spanner" required that procedural information peculiar to spanners was located with the cluster of procedures that represented "spanner".

Thirdly, the entire transition from figure 1 to figure 2 of Section 1.2.3 was concerned with spreading out procedural information in the top of the egg-timer among the data, and with adding new procedural information to particular data objects.

Finally, Section 1.3 dealt with the idea of clusters of procedures so that procedural knowledge about any one theme could be located at its corresponding cluster. We showed how a cluster provides a location for the "inside advice" associated with it.

The distribution principle will be referred to many times in the following chapters. It is applied to the distribution of knowledge about such subjects as contradictions, giving explanations and answers to "why" questions, and dialogue.

# Chapter 2 Active Descriptions

## 2.1 Introduction

The last chapter looked at some problems of representing knowledge by relational structures and came to the conclusion that both declarative and procedural knowledge about any theme should be associated together in a cluster of procedures or skills. This chapter takes up the debate and considers in more detail the relationship between procedural knowledge and declarative knowledge; that is between knowledge about how to do things or how things happened and knowledge of the static structure of things. Having clarified a few definitions we temporarily take the opposite point of view from that in Chapter 1, and give some points in favour of relational structure followed by some shortcomings of procedural representations.

We try to reconcile these two viewpoints in the last section of the chapter by suggesting a new representation which we call active descriptions. Active descriptions are structures which may be examined and hence used as descriptions, but the components of the structures are processes running in parallel where, for the moment, we define a process to be any ongoing activity to which messages can be sent and from which messages emanate. We elaborate on this definition in the next chapter. Active descriptions are like clusters of processes and develop the idea of clusters of functions presented in Chapter 1.

By a description, we mean a structure whose parts and the relationships between them give information about whatever is being described. A description is framed in some language and bears some structural similarity to the class of entities being described which makes it useful in representing and manipulating the class. A description is therefore a structure which is put to a certain type of use. If we use English as our language then our descriptions are mainly of concrete and abstract entities such as "the red top of my coffee jar", "walking fast", and "hairiness". Being descriptions they are generally represented by English noun phrases. Some noun phrases would not normally be called concepts since they are not of general enough utility.

The distinction we wish to make is between declarative descriptions, usually represented as relational structures, and procedural descriptions, usually represented by interpretable programs. Both of these are structures and both are descriptions. It is easy to confuse the issue by trying to make an incorrect distinction, as for example between structures and procedures, or by thinking that static geometrical structures in the real world may not be treated procedurally. Since this is a most important point to be clear about we will consider in more detail the particular example of the concept "chair".

A chair has a geometrical structure, since chairs are commonly made up of legs, seats, backs, and so on. One type of description of this structure is a relational structure which matches chairs and which is made of items representing parts of chairs. This is a declarative description and to use it we treat it as data for a program. Alternatively, we might have a procedural description of the static geometrical structure of a chair, perhaps in the form of a program to draw a chair with a plotter, to perceive a chair from various viewpoints, or to find the seat of a chair given any starting point.

Similarly, "chair" has certain associated knowledge which we naturally think of as procedural. Examples are "how to sit down on a chair" and "how to transport a chair from one position in a room to another". A description of either of these can be given in the form of a procedure very easily. On the other hand, much of the procedural knowledge might be omitted and replaced by declarative advice which could be used by a problem solver to form a plan for sitting down on a chair for example. This advice might be similar to "a sitting down place must be horizontal and of a reasonable size and height" or "it is not possible to sit on a chair if it is under a table". Sussman makes very similar points to these in his thesis. His program HACKER compiles procedures from declarative knowledge to enable it to manipulate blocks.

Programs, such as those written in LISP, can be considered as data to be manipulated. Similarly, data can be considered as procedures with very simple behaviour, as we show in the next chapter. The distinction we are making is between procedural content and declarative content. By procedural content we mean information about change which is explicitly

included in a form whose structure directs the flow of control.

Declarative and procedural are not separate but are two poles of a dimension. Our main purpose in this chapter is to consider the area between these poles, and suggest the kind of system which might deal equally well with both aspects.

The two poles are expressed in two paradigms from AI which are exemplified in

(1) The procedural representation of Winograd (1971).

(2) The structural descriptions of Winston (1970).

In his program, SHRDLU, Winograd treated meanings as programs. Each word and each grammatical unit had associated with it semantic programs which built up meanings for English input. These meanings were PLANNER procedures which when evaluated would answer questions or take commands. The meaning of a noun group, such as "a yellow pyramid in a box", was a program to find instances of that noun group in the program's model blocks world. So, a concept was represented by one of its uses.

Winston's program was based upon the use of structural descriptions to represent the structure of concepts from a blocks world. Such concepts were "arch", "tower", "bench" and other block constructions. A description of a concept was a relational structure which represented both the structures which should be present in an example of the concept and those which must be absent. The difficulty with relational structures is in representing the procedural information about concepts. We discussed the problems in Chapter 1 and showed that any sufficiently general interpreter for the types of relational structures needed would require so much procedural information in the structures that it would resemble a programming language.

We must try to reconcile the two viewpoints. The advantage of procedural embedding should not be thrown away, since to use concepts easily they should be programs and it turns out that we need structures which sometimes

act as programs and sometimes as descriptions.

## 2.2 Reasons for needing descriptions

In Chapter 1 we argued that it is difficult to represent procedural information using relational structures. However, in this section we argue in favour of relational structures rather than programs in cases where a description of an object is needed rather than an object satisfying that description. We will show how a description considered as a whole has similarities to the idea of "World directed invocation" expressed by Hewitt et al (1973).

### 2.2.1 Referential opacity

We need descriptions rather than programs to cope with referential opacity. A noun group is referentially opaque if a clause refers to it as a description rather than to some items denoted by that description. So in

"Does Fred know the blocks which are on BLOCK2?"

the noun group "the blocks which are on BLOCK2" should not be evaluated as in the Winograd paradigm to produce certain blocks, say B1 and B2. This would leave the sentence equivalent to

"Does Fred know B1 and B2?"

which has a different meaning. Indeed, if one person asked another the former question and neither knew the particular blocks being referred to, then the noun group could only be represented as a description.

In the case

"A thinks BLOCKA is a cube"

it might well be the case that BLOCKA is not a cube so we certainly could not represent this as the assertion

[ A thinks X ]

where X is the assertion [BLOCKA is cube] since X may be false and must not be allowed in the data-base. We would agree that the program to find the blocks may be used as the description. Any structure is a description if it is used as such. Our question is simply whether a program in a good form for finding examples is in a suitable form for other purposes.

There is a case in Winograd's program, SHRDLU, where the meaning of a noun group, although a program, is used as a description. An example is

"Pick up a big, red block"

Here the noun group is a program to find big red blocks one by one. "Pick up", however, is represented by the procedure TOPICKUP, which uses the noun group as a structure and examines it so that it first finds big red blocks which are favourable for picking up (for example, blocks with nothing on them).

### 2.2.2 Making simple assertions

A second reason for having descriptions is for making simple assertions. Winograd mentions that to deal with the assimilation of new knowledge we need improved representations of knowledge since, in general, new knowledge is not simply stored but is processed and causes alterations in the knowledge structure, sometimes of far-reaching proportions. However, even storing simple facts provides examples of our point here, that when a concept is needed for more than one purpose it might be better to have a common description worked upon by two programs rather than two separate programs each embedding the description in different ways.

SHRDLU's programs representing noun groups are composed of three types of information linked together in an order which could be thought of as a fourth type. The three are :-

(1) A set of patterns expressing factors in the description of the noun group, e.g. <<RED X>>, <<BLOCK X>>,

(2) The insertion of GOAL in front of these patterns to turn them into PLANNER statements for retrieving and testing.

(3) Statements like FIND, expressing knowledge of the part quantifiers and numerals play in the actions of retrieving and testing. For example, FIND is used in the noun group "three blocks" to find three.

In order to assert simple facts such as

"A1 is a big red block"

we would at least need to alter the occurences of "goal" making them "assert". This would work so long as we had no numerals or quantifiers. In place of the sequence of goals

```
GOAL << X BLOCK>>
GOAL << X BIG>>
GOAL << X RED>>
```

we would have the sequence of assertions

```
ASSERT << X BLOCK>>
ASSERT << X BIG>>
ASSERT << X RED>>
```

It is clearer to have the meaning of "a big red block" as a description, with "goal" or "assert" being given as an argument, as in

```
ARG << X BLOCK>>
ARG << X BIG>>
ARG << X RED>>
```

Later we show that the best way to find an instance of a set of patterns is not necessarily to find each one in some sequence. So we could even take out "arg" from each pattern and simply say "arg" of the entire description as in

ARG [ <<X BLOCK>>
<<X BIG>>
<<X RED>> ]

The set of patterns is now very like a Winston structural description.

If we think of the individual patterns as processes connected by way of "X" then the set of patterns corresponds in some ways to a cluster of procedures as described in Chapter 1. The procedures associated with <<X BLOCK>> might be to find examples of blocks and to test objects for blockness. They may be processes rather than procedures since the block instance generator may produce examples one at a time and remember its position in the set of examples.

The patterns considered together as a group also relate to the "world directed invocation" mentioned in Hewitt (1973) where a group of patterns is considered as a micro-world with processing ability of its own and corresponding to the meaning of the group.

### 2.2.3 Assertion and referential opacity

The next example relates directly to the previous ones. A noun group sometimes refers to the entire description as a microworld rather than to some particular part of the description. Suppose we say (rather simplisticly)

"My piano is a box with strings in"

then we do not really want the separate assertions

"My piano is a box"
"The box has strings in".

Instead we want the piano to be the entire object, box and strings together. In other words, rather than "find a box", "check it has strings in", "match the box with the piano", we want to "match the

piano to an instantiation of the entire description "box with strings in". We need an instance of the concept "piano" to represent the piano as a whole.

This shows up a difference between Winston's method of structural descriptions and Winograd's procedural representation. In Winston's work an "arch" could be defined as a description

"an arch is a beam with two supporting blocks".

In Winograd's program the definition would be a program

"find a beam and check it has two supports".

So Winograd's system would say B1 is an arch in figure 3 and this is non-intuitive.

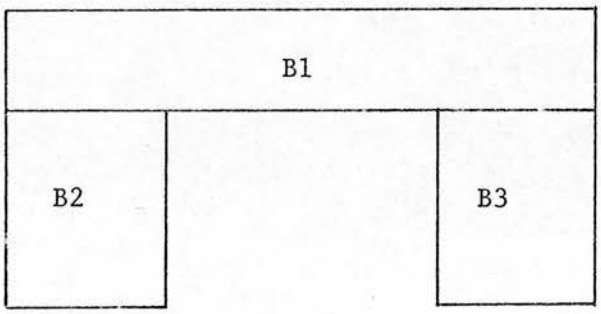



Figure 3

2.2.4 Numerals

If we only use noun groups in sentences where examples of the noun groups are needed then numerals are no problem. So, "put two blocks in the box" would use "two", in the program corresponding to "two blocks", to count the examples of blocks as they are found. On the other hand, suppose we want to assert about a blocks world the statement "two blocks are in the box". If the program cannot see inside the box and cannot work out which blocks they are, all it is able to do is to store that something satisfying the description "two blocks" is in the box.

The same problem occurs when we represent hypothetical actions as in

> "If I put two blocks from the table on to the shelf how many red blocks would be on the shelf?"

where there are no blocks on the shelf to begin with and, say, all blocks on the table are red. We certainly do not want to run the "put" program twice, moving two particular blocks, and then to assume that since these two happen to be red we will always end up with two red blocks on the shelf. If there had been a purple block on the table this might be false. We need to reason about the action and, in so doing, to manipulate a description of it if we are to arrive at a true result. We need to do something akin to symbolic evaluation as used in proving correctness of programs.

## 2.3 Difficulties in procedural representations

So far in this chapter we have discussed the good points of examinable relational structures. We now argue that current procedural representations have shortcomings. In particular we consider the mechanism of PLANNER type deduction for finding instantiations of sets of patterns, for asserting objects to be examples of concepts, and for reasoning about concepts in general. Some of the arguments will be against preferred uses of PLANNER rather than about what is possible using it.

Let us take as a first approximation to a description a set of patterns with common variables to be instantiated. This could be seen as a local self-contained data-base. The important points about this scheme are

(i) The description is not accessed from any particular starting point.

(ii) There are links between the patterns.

(iii) The patterns are in no particular order.

Figure 4 shows a possible definition of an "arch".

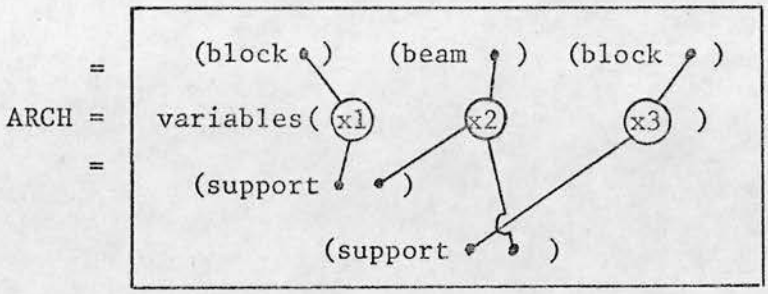



Figure 4

In PLANNER a natural way of making such a concept from smaller ones would be to choose a particular ordering of the patterns, envelop each in a goal statement and make the entire structure a consequent procedure for finding and testing examples of the new concept.

There is another natural use of PLANNER for representing concepts such as the arch of figure 4. This is to make an assertion in the data-base for each pattern at any new instance of the concept. An advantage of this method is that we can express relationships between the individual assertions and do some extra deductions to fill out the picture of the instance as it is constructed. We embed the knowledge for this in antecedent theorems related to individual patterns which are invoked when the patterns are asserted. These theorems represent the concept.

However, both approaches are open to criticism. Those concerning the first method are based on faults in the ordering of individual patterns which must be made if the patterns are to be combined into a single procedure. They are too tightly combined. We criticise the second approach because the individual antecedent theorems and patterns are too loosely combined. We also show that both methods require sophisticated communication between the patterns.

### 2.3.1 Problems with the first method

2.3.1.1 Choice of an order for the patterns

Firstly, we shall consider the choice of an order and the problems entailed by having to make one. SHRDLU has a very simple method of dealing with this problem. When building the meaning of a noun group, it orders the patterns according to weights associated with them by the programmer. This is adequate for the small blocks world used. However, for a larger world things can go wrong. There are two problems. One is in the method used to order the predicates, and the second is in the time this ordering is done, e.g. how do we know that fixed weights will always work? The weights might depend on the context at the time of ordering. Certainly the choice between second and third placings might well depend on the choice for the first pattern to be goaled. Even worse is the possibility that order of execution might depend on context at the time of execution. "Look for a linguist who plays the tuba" is best done in one order at a linguistics conference and another at a brass band competition.

2.3.1.2 Problems when the patterns are already ordered

Even given a particular ordering of patterns, things can go wrong. Suppose we try to find an "Egyptian Elephant" using the sequence of goals

```
GOAL<<EGYPTIAN £>X>>
GOAL<<ELEPHANT £>X>>
```

How is it to be discovered that these goals are inconsistent given that Elephants are either African or Indian. Suppose the first goal discovers an Egyptian object E1. The second goal will fail and another Egyptian object will be searched for. The mistake is made because the two goals are considered independently. The second goal might instead return a reason for failure saying that Elephants cannot be Egyptian. This can only be used if the second goal knows that £>X is Egyptian because of the first goal. In this case, if it knows the structure of the program it is in, when it fails it can give a failure message as a reason.

There seems to be no simple general way of arranging that the best advantage is always taken of possible interactions between goals without

abandoning the idea of a particular ordering of goals. Any statement needs to know not only the structure of its immediate textually containing theorems, but also the structure of deductions made at run-time. A variation of our example will illustrate this. Consider now

```
GOAL<<FOREIGN £>X>>
GOAL<<ELEPHANT £>X>>
```

The first goal could cause various computations in the course of which <<EGYPTIAN £>X>> might be asserted or hypothesised. GOAL<<ELEPHANT £>X>> would fail because of this assertion. To allow for this, GOAL<<ELEPHANT £>X>> must be able to see what part<<EGYPTIAN £>X>> played in the execution of GOAL<<FOREIGN £>X>> and what conditions implied it. Having found this out, appropriate action can be taken to prevent <<EGYPTIAN £>X>> being tried again.

Another approach to the problems, besides letting <<ELEPHANT £>X>> do the work of tracing the difficulty, is to let it give advice to GOAL<<FOREIGN £>X>>, namely, "don't find anything which has <<EGYPTIAN £>X>> true about it". How GOAL<<FOREIGN £>X>> uses this advice is a difficult question to answer in general, so the decision should be made by the object representing GOAL<<FOREIGN £>X>>. Allowing communications of that sort to goal statements moves towards the approach we advocate in Section 2.4 where individual patterns themselves are processes taking sophisticated messages as arguments.

2.3.1.3 No order for the patterns might be satisfactory

A third problem that arises when we proceduralise the description in a PLANNER-like way is that there may be no satisfactory order. Consider the case "a point on line1 and on line2" in figure 5 where line1 and line2 are circles centre O and P respectively.

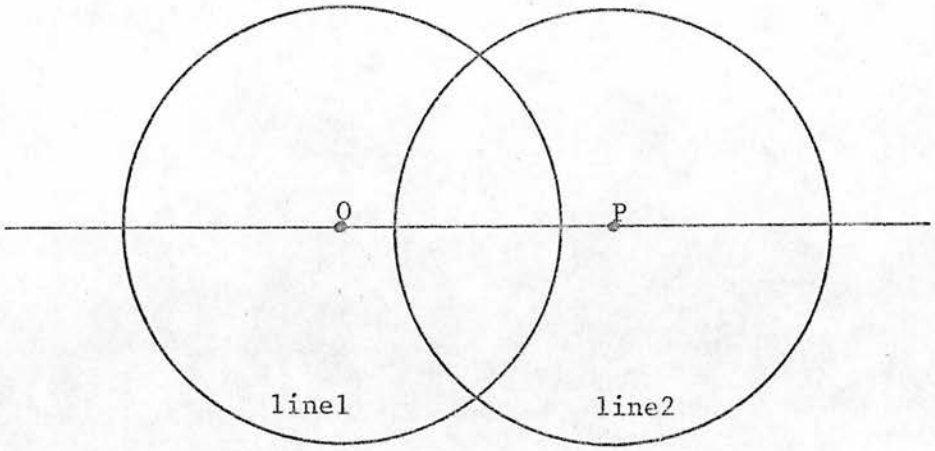



Figure 5

To try to satisfy these two goals independently in either order is foolhardy since there are an infinite number of solutions to each but only two to them both. Instead of simply searching, some deduction needs to be done first. It is the situation of figure 6 where we know our answer is in set A and in set B and where together they imply it is in the smaller set C.

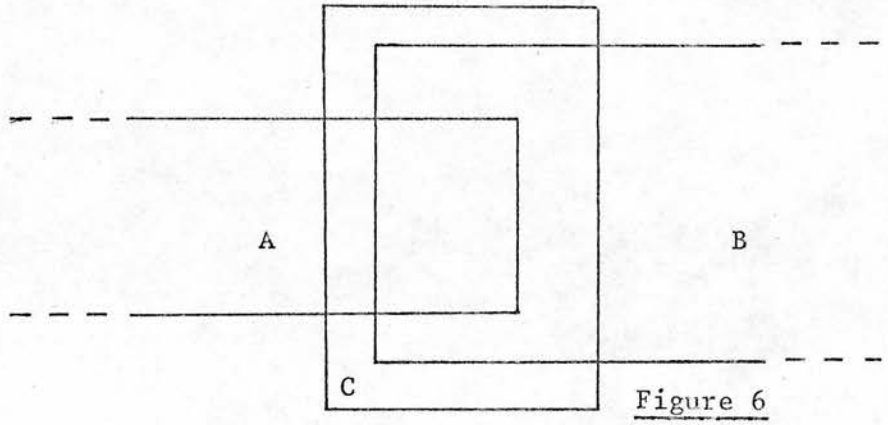



Figure 6

We can determine the set C by forward reasoning from the given information. This means that given certain information about a situation reasonable deductions are immediately made to expand on the picture of the situation in hand. The gaps which are filled in and the deductions that are done will generally be dependant on a classification of the situation and so is context specific knowledge. Forward reasoning is related to perception since known information acts as cues for desired information.

### 2.3.1.4 PLANNER retrieval relies too heavily on central data-bases

A final criticism of the PLANNER type approach to concepts is that it is too oriented to the use of a central data-base and the concept of retrieval from such a data-base as a simple, perhaps externally directed search (internal

and external advice were discussed in Section 1.3). Procedures can be used to express implicit information, but problems so far considered have not been real-world enough to require very much procedural information about how to use procedures. This leaves programs much too tied to a central data-base of knowledge rather than well-organised knowledge.

Let us take two simple cases. Firstly, consider "find a lion in London". Instead of looking at all lions in the data-base and seeing if they are in London, it is more sensible to realise that if the lion is to be in a city it will be in a zoo or a circus, and then to look for zoos. Having found a zoo we can use the knowledge we have about zoos and search for a map on a bill-board. Using our knowledge about maps, we are home. If we were in a jungle, it might be wise to employ a tracker with refined methods for finding lions. We can treat London, zoos, maps and the tracker as data-bases.

Secondly, I could ask for "a cup with a chip on it". Now if my kitchen were impeccably kept it might, on data-base consideration, be wise to look for something chipped first as there will be fewer of these than of cups. In the real world though, it is better to look for cups since we know where to find them. We would look in the kitchen cabinet which is like a data-base itself.

A data-base should be like a shop with a storekeeper. If we want something we ask the storekeeper who knows how his store is organised. He can give us advice by conversation about what we might need for any particular purpose and he might use reasoning to find this advice. If he were a chemist for example, he might diagnose that we had a common cold and prescribe something to alleviate discomfort. Using our definitions from Chapter 1, we can say that our knowledge of the world should be organised from an internal viewpoint. This reduces the importance of retrieval from a data-base and emphasises the role of deductions.

We know of no program which behaves like the storekeeper in this example and suggest that it would make an interesting project.

### 2.3.2 Problems with the second method

We have considered problems involved in representing descriptions as PLANNER consequent procedures for finding examples so let us now examine the second method of representation.

We must consider what happens when a description is originally built. Taking as a starting point Winograd's system, let us examine how SHRDLU puts together its noun group meanings. Each noun or adjective has a meaning which is a program, and the majority of these add patterns onto the PLANNER procedure for the noun group checking for compatibility between the new part and the procedure being augmented. In some cases only one addition is made, as in "block", which causes [goal [block f>X]] to be added. Other cases add more than one statement, for instance "cube" adds one statement to find a block and one to check that its dimensions are equal. Certain words even slightly alter the patterns already present. The semantic consistency checks which are made are very simple checks that certain semantic markers do not conflict. In both addition to a description and in checking consistency, however, deduction has a major role to play.

A natural way to achieve this in PLANNER is to represent the parts of a description by patterns in a data-base. We can then use antecedent theorems to represent knowledge about building descriptions and use consequent theorems to represent knowledge about finding instances of descriptions. We consider a noun group to be a set of assertions about an object represented by a variable. So in the case "Find a big red block" we have an atom, CONCEPT1, representing the concept expressed in the noun phrase, and in building up the noun group we assert the patterns

```
<<BLOCK CONCEPT1>>
<<BIG CONCEPT1>>
<<RED CONCEPT1>>
```

This collection of assertions is supposed to represent the meaning of the

noun group CONCEPT1 but the assertions are separate. However, we do have the advantages of antecedent theorems when we build such a description. The simple semantic marker check between "big" and "block" can be programmed as an antecedent theorem invoked when we assert that the concept is big. It checks that the other assertions are consistent with "big". More complex checks and also deductions adding to the description can be programmed by antecedent theorems. So, "A Newfoundlander" would cause the assertion <<NEWFOUNDLANDER CONCEPT1>> which could invoke antecedent theorems asserting <<PERSON CONCEPT1>> and <<LIVE CONCEPT1 NEWFOUNDLAND>> and deleting <<NEWFOUNDLANDER CONCEPT1>>. Knowledge of semantics of words is thus separable into many antecedent theorems.

If we say "find a CONCEPT1" then "find" has to search through all the data-base to collect up the assertions concerning CONCEPT1 before it can begin a search for an instance. To prevent this we really need a separate data-base for each concept containing the assertions about it. While this is being used to find an instance of the concept the atom CONCEPT1 is treated as a variable which is instantiated in various ways. The description of the concept looks rather like a PLANNER procedure again so we have come around full circle. We now know that building a PLANNER procedure can be thought of as making assertions in a data-base. The assertions can invoke the use of knowledge about special relations between these assertions. In other words, procedure-writing knowledge is being expressed as antecedent theorems which watch over the building of the procedure. The program is being treated as data and is in the data-base as assertions.

We have seen that the various parts of a description have interactions and allow deductions to be made, effectively adding to the description. We have seen that this can occur at least at two times : when the description is being constructed and when it is being used to test or to instantiate. The process is one of construction by deduction. For example, if we have to find an example of some description we might not search immediately, but instead might consider the description to see what it entails. In doing so we build up a picture of the typical object we are looking for.

It seems unreasonable that such a description can only be refined by mutual implications between its parts. What about the relation between the rest of

the world model and a particular description? A new piece of knowledge added to the system might modify a description already built by allowing further implications to be made about it. In general, this is how our concepts can develop, be generalised, enhanced, simplified, etc. In particular, in language processing, interactions by deduction between two separate partially built descriptions within a currently processed sentence might direct parsing, and remove ambiguities. Our picture is of descriptions which are active at all times.

There are two other problems with basing our deduction on antecedent theorems (demons) and consequent theorems. Firstly, it seems we still have to choose an order for putting patterns in the description even if we do not decide to fasten them rigidly together into a procedure. Secondly, with too many antecedent and consequent theorems it is difficult to see how a program will behave. We can have a surfeit of demons where the main program is halted to take care of side-effects which in turn take care of more side-effects so that the main task is hardly tackled. Both of these problems are concerned with scheduling and if programs get much larger, but still are uniprocessing, an inordinate amount of code will be devoted to scheduling problems. This is a kind of exponential explosion and we show in the next section how a hierarchy of parallel processes might prevent it.

To sum up, we have considered some requirements for a system of concepts based on descriptions and analysed some problems in representation. In retrieving examples of concepts, deduction plays an important role, in particular, for expressing inter-relations between individual components of the description. It also seems that the components (i.e. individual patterns) require some degree of autonomy as processes and we next argue that they should run in parallel.

## 2.4 Proposed implementation as parallel processes

We propose that each pattern of a description be a separate process and, to ease the problems of co-ordinating them, that they should be considered as running in parallel. Any linearisation in time which is not sufficiently fine to approximate to parallelism will produce large problems of organisation. We saw in the simple case of a two-goal description how

two goals are often dependent, and how discoveries by one may help the other. The order in which these discoveries are made is extremely situation dependent so we doubt whether rules for ordering of separate processes on one processor could be expressed without either making the rules very run-time sensitive, or making the time-slice small enough to approximate parallelism.

Whatever the outcome of the argument for parallelism, there is a strong case for having each pattern be a process rather than a procedure. This is because it is so simple to construct examples where "one goal finds something out which helps a second to find something which helps the first, and so on". The goals must be processes if they are to give and receive advice.

The separate processes must have channels along which to communicate and whether these are arranged as in a telephone exchange or are fixed private lines the individual processes must know directly most of the processes they usually communicate with. Otherwise, since the number of possible connections between n processes is n*n, the work that any central administration would have to do to answer every processes' requests for connection to processes with desired properties would grow as the square of n. Moreover if the number of possible properties increased in proportion to n then the work would grow as the third power of n.

The results of processes and advice from them should be usable by certain others. We propose that this should be done by side-effecting. Consider two processes which are part of a description. One is associated with the pattern of <<A £>X>> and the other with <<B £>X>> . The entire description is asked to find an example of itself, so the separate patterns each try to find values for x which satisfy their own predicate. The variable £>x is treated as a common process so that if one process assigns to it the other can tell. It is also treated as an object since either process might need to assert something about the object. In particular, the predicates <<A £>X>> and <<B £>X>> are both asserted about £>x and the process for £>x should know both. We attach the assertion concerning £>x to £>x itself reversibly so that not only do the assertions point to £>x but also £>x points back. Then each process can ask £>x what its current

state is and f>x can notify each of the changes to itself.

Again, suppose we asserted that f>x was a car. If the assertion <<CAR f>x>> was itself a process running in parallel with the other processes then it might keep a watch on f>x. If certain other details were asserted of f>x, such as number of wheels, engine size, etc., these should "react" with the assertion "car" to produce a more detailed picture of the car. This use of "car" must, of course, be an instance of "car" as we do not want assertions about one car to be transferred to all cars.

We can use the CAR example to show why parallel processing is necessary. If we have a single processor with access to a lot of care then every time f>x is changed all the processes which represent assertions about it must be run just in case one of them needs to react. Although we might know that only a few will react we do not know which since that knowledge is best associated with the particular process. So we spend a lot of time checking demon processes. On the other hand the demon processes may need only limited processing power and almost certainly need a very limited store and limited communication channels. This would ideally suit a machine with many small connected processors, and such a machine is easily imaginable. There is every reason to believe that the human brain is such a machine since to denigrate most of it to the function of passive storage when a thousand processors can be grown as easily as one seems to be taking the more unlikely standpoint.

The idea of concepts "reacting", which we mentioned above, is another example of the need for parallel processes as opposed to roughly interleaved processes. Consider a student and teaching program reaching via a console. The program has control, asks a question and hands over control to the student. If the program must halt until the student replies than it cannot spend time thinking ahead. Even worse it cannot interrupt if the student spends too long thinking. Either a fine time-slicing independent of the program structure is needed or else true parallelism.

A limiting factor on the size of the AI programs has been our ability to properly represent concepts as processes with declarative and procedural knowledge and with aquaintances which are other processes with which they can communicate. If such a representation were perfected then our programs

could be hierarchically organised as suggested in Chapter 1. We could have clusters of small communicating processes and some members of these would communicate with some members of other clusters, thus grouping clusters into larger clusters of clusters, and so on. All the lowest level of clusters would have a limited computing power and would operate in parallel. This scheme would be of great benefit in preventing combinatorial explosions.

For suppose each cluster has m subclusters and there are n levels in the hierarchy so that the total number of base level processes is $m^n = N$. Now if each cluster may communicate with any other cluster provided it is within the same supercluster then the total number of possible communications channels needed for any cluster is $m^2$. But the number of clusters at the ith level from the top is $m^i$. So the total number of channels needed at that level is $m^{i+2}$. Thus the grand total of channels needed is

$$m^2(m^n + m^{n-1} \ldots\ldots + 1)$$

$$= \frac{m^2(m^n - 1)}{m - 1} = \frac{m^2(N - 1)}{m - 1}$$

which is only of the order of N. In the unstructured case, the total number of channels needed is $N^2$ which is explosive.

Moreover, there is no need to have such a strict hierarchy since we only introduced this to simplify the calculation. Our demand is only that the range of any process or cluster of processes is limited. We could, for example, use a base process to stand for a cluster of clusters whose external behaviour was simple.

Until programs can construct large units in the way we describe, programs covering large bodies of knowledge will be messy and will suffer from disastrous interactions. Concept building is possible if concepts are considered as relational structures and their working substructures. However, the fault with these is that they are difficult to interpret. Programs are far more flexibly interpreted but have certain restrictions for use as concepts. The answer lies towards active descriptions and distribution of knowledge. To allow clusters of expertise to be amalga-

mated, multi-processing, and more meaningful inter-process communication are necessary. This can lead to many anthropomorphisms. The idea of many processes simultaneously operating, giving each other advice, conversing and side-effecting an environment, which is itself only a collection of processes, is extremely suggestive. It can be dangerous if our recursion is not based on anything and we put a homunculus into each process. On the other hand it can be an endless source of ideas for computational possibilities.

## Chapter 3 The Structure of processes

### 3.1 Introduction and definition

A function is characterised by a fixed input-output mapping. This implies that a program to model a human being cannot be a function except in an obscure way, since the important property of complicated behaviour is its structure and not its end product. Ways of structuring behaviour and the interactions or side-effects of one part of a system on another become more significant than a single input-output relation. We consider that intelligent behaviour is made up of processes and that even simple behaviours are more easily represented in terms of processes than in terms of functions. In this chapter we will show what we mean by a process and why it is advantageous to think of behaviours as processes. Throughout this thesis we describe simple programs which are processy in special purpose ways. P-74 is an attempt to provide a programming language enabling us to surmount the obstacles inherent in functional viewpoints of programming. Certain chapters, notably Chapter 2, propose systems based on many interacting parallel processes. Here we try to distinguish some conceptual problems about processes which affect our theory of knowledge representation.

> Definition: We define a process to be any ongoing activity which can be acted upon and which acts upon its environment.

So, a car standing idly by the side of the road is a process. We can act upon it by pushing from behind and it will respond by moving down the road. As a process it must be composed of other processes, such as the wheels which turn as we push. Something else might interact with one of these sub-processes, for instance the brake. This could make the total process respond to our push in a different way. The process is ongoing, so some parts of it might suddenly give an output without having any new input. The axles could collapse, for example. Generally in our context, the activity of the process is some computing activity. In this case there need not be a material object apparently corresponding to the activity,

as there seemed to be in the case of the car. Consider "the turning of the wheel" as the object to be represented rather than "the wheel" and some flavour of our approach will be there.

Notice that this definition of process also includes the execution of functions. With these the only important interactions are at the beginning and end of the function execution unless the function or any of its inner functions have side-effects.

A process has a very important property called its internal state. Whereas the state of a function is the same after each execution, this is not so for processes in general. An input may alter the state so that the next input produces a different response. In this respect a process can be modelled by an automaton in its technical sense of a set of discrete states with mappings describing state changes and outputs for any input and current state. One way in which general processes differ from automata is in expressibility. The difference is like that between a Turing machine and a high-level language. An automaton is generally given one of a set of discrete uncoded, unstructured inputs and this produces a transition from one named, unstructured state to another.

Although POP-2 is designed as a functional language, processes other than functions do show up as character repeaters, and functions with non-local storage attached either at compile time or by means of closures. This is not surprising when we remember that functional treatments are unconcerned about side-effects. Since reading and writing characters involve side-effects, it is natural that they should require processes. Consider a function which outputs characters on a teletype. It keeps track of the number of characters it has output and if it outputs more than 80 after any new lines, it outputs an extra new line. It is a process whose internal state records the number of characters output since the last new line.

The second part of this thesis includes a description of a control structure we developed, called P-74. After experimenting with P-74, we discovered that a key part was important in itself. This is the ability to produce an object called a process which can be given a message and set into action, and which can in turn set off a similar object, perhaps giving that object

a continuation point just in case the new process ever wishes to return control. The code for this ability is only a very small part of P-74. We built P-74 around it, though we could equally have programmed other control structures. In particular we could write iterative or recursive control structures. For iterative structures we represent instructions as processes which do the same activity every time they are called. By giving any instance of an instruction a reference to the next instruction in sequence we can produce sequential code, and by giving instances of the special "goto" and "conditional" processes' references to their relevant next instructions, we can produce flowcharts. It is almost as easy to implement recursion. We simply define processes which can be given one of two messages : either "I am returning to you", or "I am calling you". The calling message will include a return process and argument; and the return message will include results. If we wish to implement local variables then the message can include an environment.

Figure 7 shows how the interpretation of flowcharts is only a special case of the use of processes.

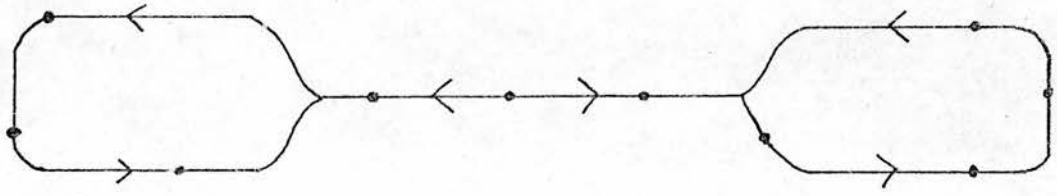


Figure 7

The two looped line represents a road which is used as a bus route. The bus follows the arrows and encounters a series of bus stops represented by the large dots. We consider the bus to be a process, part of whose state is its position on the route, its direction and its passengers. At any bus stop the bus checks if any passengers need picking up or setting down, executes the operation, and moves on to the next stop. We could make the passengers themselves into processes who are moved along by the bus and who request the bus to stop when necessary. We can also relate it to the interpretation of a flowchart with the bus route as flowchart and bus stops as instructions. The bus would then be the interpreter. Our example clearly shows that processes are more general than flowchart interpreting.

For one thing the bus goes both ways along part of the flowchart. For another, the passengers switch from being instructions at bus stops to part of the interpreter in the bus.

Our key part of P-74 resembles the primitive "actor transmission" of Hewitt et al (1973). Hewitt puts forward actor transmission as a universal primitive and says that iteration, recursion, and backtrack are covered by it. The point is that we should not restrict ourselves to any one control structure but should, by thinking of processes, allow all and any to be used. The organisation of an intelligent process should be domain dependent and therefore vary over the parts of the process. It is wrong to squeeze it into any uniform control structure except at a very primitive level. Wittgenstein (1953) made a similar observation about language :

> "Our language can be seen as an ancient city : a maze of little streets and squares, of old and new houses, and of houses with additions from various periods; and this surrounded by a multitude of new boroughs with straight regular streets and uniform houses."

## 3.2 Processes as building bricks

We can use the quotation from Wittgenstein to make an observation of our own :

> "The science of artificial intelligence will be more a science of town planning than a science of bricks or geometry. It will be a melange of differing styles of theory interacting in subtle ways."

This partly answers a common criticism of the "community of processes" paradigm of Minsky and Papert (1972), namely that a system with no uniform mathematical structure as a framework will inevitably be disorganised, and that we should therefore search for such frameworks and see our problems in terms of them. We say such a system need not be unstructured simply because it is not uniformly structured. The structure will depend on the problem and we know that patterns found in the real world, both natural and man-made, are extremely various. There is still a problem. Relegating actors and processes to the role or primitive building blocks gives us

freedom, but in doing so does not direct us in planning our programming of large projects. To overcome this we need to invoke the "split the problem into parts" dictum. Processes must be decomposable into combinations of sub-processes and combinable into larger processes. This is not as easy as it sounds. We thought of a process as some object to which we could send messages. If we combine processes by joining inputs and outputs, making a cluster in which certain processes can contact others, we do not have an object to represent any new process. Moreover, the collection of processes has many inputs and outputs whereas our original processes had only one way of receiving a message. This situation is illustrated in Figure 8. Here the conglomerate

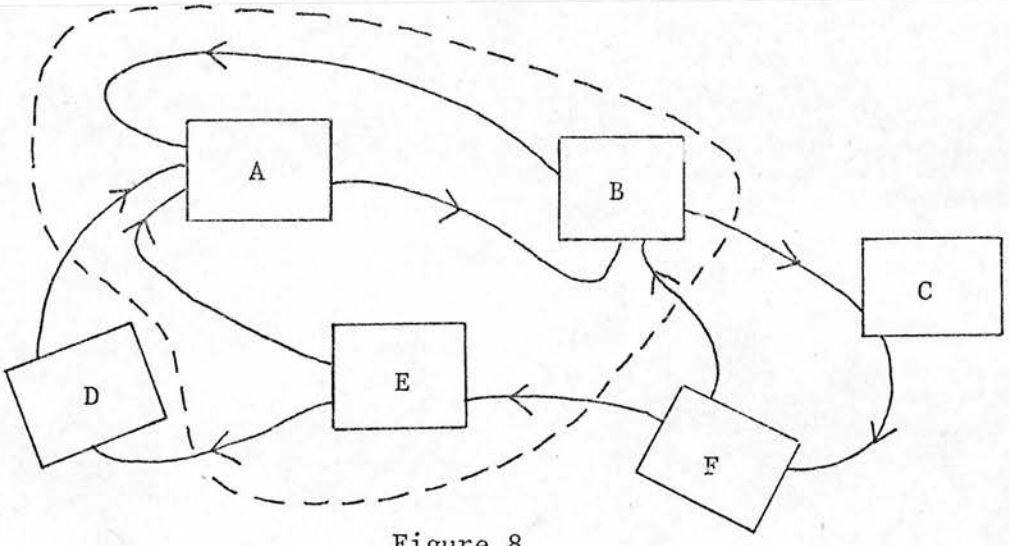



Figure 8

process within the dotted lines has three different inputs. In what sense can F be considered as communicating with the big process rather than with either E or B ? If we try to rectify this situation by making new objects representing process combinations then we have telephone exchange problems in sending incoming calls to the relevant component. We must also allow an outside process to know about an inside component.

The analogy with a living cell is appropriate here. Every cell has many parts of different kinds and with different functions which in some way combine to make up the total behaviour of a cell. In their turn, the cells themselves are combined to make larger organs. Each cell however has a nucleus which coordinates the behaviour of the cell parts although the parts

have some independence and can receive messages from outside the cell or from other parts of the cell. It is significant that a cell has a membrane.

We can go in the other direction and decompose primitive processes. Any of the boxes in Figure 8 could be thought of as a collection of smaller processes with inputs and outputs connected. We have shown (1972) how a close look at even a primitive function like "assign" reveals a complex structure. If we had storage locations A and B which can only hold 0 or 1 then assign A to B becomes

```
IF THE CONTENTS OF A ARE 0
THEN IF STORE B IS 0
     THEN
     ELSE ALTER STOREB TO 0
     CLOSE
ELSE IF STORE B IS 0
     THEN ALTER STORE B TO 1
     ELSE
     CLOSE
CLOSE;
```

It seems there is no fundamental type of building block. We have just a set of descriptive methods.

In computing terminology we tend to visualise a process as a piece of code which is being interpreted by a processor. This is sometimes a restricting approach and the alternative of considering a process as an interacting cluster of sub-processes is often more pertinent. It is quite possible for such a cluster to behave as though it were a simple interpreted algorithm some of the time and at other times to behave in another related way.

A simple example will help to illustrate this. The well-known game of "Life" (Conway, 1970) is a rectangular cellular automaton with cells that have possible states and which may change state at the end of each unit time interval according to rules determined by the states of their immediate neighbours. The purpose of the game is to watch patterns of "on" states change as time passes. In this situation we have a clear case of many small parallel processes interacting. For some settings of these processes it is possible to describe what is happening on the screen

in terms of much larger processes, such as "a traffic light blinking", "a glider moving across the screen". Although we could write processes to simulate these behaviours there is nothing in the cellular automaton which we can point to as the processor for them. They are artifacts, and maybe many processes are artifacts in the same sense. It is mistaken to represent this process as a piece of code which is being interpreted. Such code can easily explain the behaviour of one glider in an empty space but when it collides with other patterns the only simple description of its behaviour is in terms of the behaviour of its parts. The same problem occurs repeatedly. In planning situations, the relationship between a high level plan and its detailed version follows this pattern. It may be difficult to decompose the detailed plan into separate larger units although during planning the larger units were of great assistance. When the units are juxtaposed there are many interaction bugs which need special purpose local patching.

Sussman's HACKER (1973) partly tackled this problem in the domain of making plans for block manipulations. Since HACKER incorporates a programmer, HACKER programs are continually changing. Consequently it is difficult to draw the line bounding a program; some later execution might add to it and draw upon knowledge diffused through the system.

However the problems of combining processes are resolved, we agree with the quote from Ross (Hewitt, et al, 1973).

> "Our primary thesis is that there can and must exist a single language for software engineering which is usable at all stages of design from the initial conception through to the final stage in which the last bit is solidly in place on some hardware computing system."
>
> (Ross, 1973)

> "This illustrates our belief that methods of structuring processes that are discovered will be applicable at all levels of unit."
>
> (Stansfield, 1972)

Our approach to combinations of processes leads us away from languages like SIMULA (Dahl and Nygaard, 1966) which also allow many parallel

processes to be coordinated. We suggest that each parallel process should itself by structured from processes. The processes of SIMULA are related to co-routines. They are structured as ALGOL recursive programs with the usual hierarchical block structure and with each block being a flowchart. We discuss this point in later chapters which give a technical review of many control regimes and describe how our regime, P-74, generalises these.

## 3.3 Intensionality

The relationship between a function and its arguments is more complex than one might think and cannot always be described in terms of program versus data. Sometimes a program's behaviour may be almost independent of its arguments, in other cases it may interpret them and yet again it may treat its arguments as procedures themselves. At one extreme we have a program which takes a number and does complex manipulations on it to produce a result. The behaviour of this program through time and the structure of its flow of control might depend entirely on the program. At the other extreme we have a program such as a LISP interpreter whose behaviour is so dependent on its argument that this argument is also termed a program. There are myriads of in-between cases. It seems far better to forget the distinction between program and data and consider simply interactions between processes. Data itself becomes process with a certain simple behaviour.

We suggested in Chapter 2 that patterns should be treated as active processes and we now show how a smooth transition can be made from one to the other. A pattern is essentially a data-structure consisting of several linked boxes which can be filled with items and accessed to see what items they contain. Usually, such record-structures are represented as static items to be operated on, rather than as programs. Hewitt showed how his actor formalism allowed a list-cell data-structure to be represented so that the list cell itself took the responsibility for processing. Instead of providing functions CAR and CDR for updating and accessing a static list cell, the list cell was an actor which could be given various messages. These would cause it to behave in different ways. The actor could be told to give the value of its CAR or could be told to absorb a new value for its CAR, for example. In other words the

roles of function and argument have been reversed. We could represent a list cell as a process in POP-2 by using partial application as follows

```
FUNCTION CONS A B;  VARS SELF;
LAMBDA MESS A B SELF;
IF MESS = "CAR" THEN A EXIT;
IF MESS = "CDR" THEN B EXIT;
IF MESS = "NEWCAR" THEN->FROZVAL (1, SELF) EXIT;
IF MESS = "NEWCDR" THEN->FROZVAL (2, SELF) EXIT;
END (% A, B, 0, %)-> SELF;
SELF->FROZVAL (3, SELF);
SELF;
END;
```

CONS is a function which takes two arguments and which produces a closure of the function LAMBDA ... END; with A, B and SELF initialized to the arguments of CONS and the new closure itself. Thus, the closure may alter its initial values by assigning an argument from the stack to the relevant frozen value N. It does this by

```
-> FROZVAL (<N>, SELF);
```

CONS(1,2) results in a function which has an internal state frozen in. In other words a simple process.

We next give some examples which show why the representation of data as processing gives us greater freedom.

Firstly, we have experimented with processes as a representation for the semantic structure of sentences. Our representation is based upon verbs, each verb being associated with some process. As any clause is parsed, sentence fragments are given to the verb, together with messages which help the verb to allocate these. For example, the object of the clause might be given to the verb to store and relate to other clause fragments. We could use a slot and filler representation regarding a verb as a structure with spaces corresponding to the various participants and circumstances, and altogether representing some event denoted by the verb. However, that would require the function which fragments clauses to know about all the possibilities and problems for allocating fragments. It would have to know the semantics of all verbs. It is much clearer for each verb

to know about its own semantics. This is an area worth much more investigation.

Although a variable is normally regarded as a passive box into which we put objects, we can make active ones as follows

```
FUNCTION NEWVARIABLE VALUE;  VARS SELF;
LAMBDA A SELF;  VARS MESS;
IF MESS = "IN" THEN->FROZVAL (1, SELF) EXIT,
IF MESS = "OUT" THEN A EXIT
END (% UNDEF, 0 %)->SELF;
SELF->FROZVAL (2, SELF); SELF
END;
```

Since the variable now takes over control whenever it is used we can give it responsibility for other actions. For example, we could make a particular variable refuse to accept anything which had the property "square". The way to do this without making the variable active would be to alter every piece of code which assigned to the variable. Alternatively we could alter the code of the "assign" function itself but as we might need this for many variables it would become inefficient.

This property of data-structure definitions, which allows the structures to take on the responsibility for actions involving themselves, is called "intensionality" by Kay (Hewitt et al, 1973). The idea has been around in various forms for some time. It is a feature of the language GEDANKEN (Reynolds, 1970) that many of its data structures are functional. Also the record structures attributed to Hoare (1972) are intensionally defined to some extent.

We normally think of real world objects as being passive and therefore with extensionally defined behaviour. We think of what we want to do with them. It is easy to find examples contrary to this where intensional definitions are more appropriate. Suppose we pick up a box. It may have something heavy in it or else may contain a light object. The response of the box to the force we apply will be dependent on the weight and it might be easier to have the box itself know how to respond. If the box is large and has a heavy ball inside it which rolls around it would be easier to

define the box intensionally. Finally, consider if the box contained a bomb or a gyroscope.

Intensional definitions are especially useful in the presence of parallelism. Suppose we had a simple electronic circuit with several inputs which could be operating simultaneously. If we defined the circuit extensionally then every input would have to examine every other to decide what state changes it should make to the circuit. This function should not be duplicated but should be programmed as part of the circuit's own behaviour.

When introducing a new data type, we should think first of all about its behaviour, about what it does when it interacts with external processes. The reader will recall the method of building a new cluster of skills about a particular theme, as described in Chapter 1. In that case also we asked what behaviour needed to be associated with the theme and what orders the new cluster should be able to obey. In the present case we have a choice for representing this behaviour : either to write a set of functions to act on data structures, or to make each data item react to a set of messages. The first branch of this choice corresponds to extensional definition and the second to intensional. Hewitt extols the virtues of the intensional approach and indeed we have just given examples where it has important advantages. Since it implies that we have no direct access to the structure of a data-item but only have access to its behaviour, this means that the precise implementation of any data-item is of no concern. We can alter one or two data items by giving them more expertise, but as long as they include the old behaviour they will be acceptable members of the data class.

Unfortunately, a simple example will show the issue is not quite so clear-cut. Suppose we required a program to double integers. We could consider, extensionally, that "double" was a function whereas integers were objects. Intensionally we could make "double" simply give a message to a number and make each number know how to add itself onto itself. The intensional definition allows us to extend integers to reals, negatives, fractions and so on without modifying double at all. The new types of number must simply respond correctly to the message "double". Double will still work perfectly

for integers. We would have more difficulty extending our program to cope with an intensional version of "treble" since it would need to send a type of message the numbers are not prepared to deal with. Either "treble" must work in terms of the messages that numbers expect or else the behaviour of numbers must be augmented.

It seems that intensionality and extensionality are really complementary. In the case of the extensional definition of "double", the "double" function would at some level ask questions of its argument. It might ask for the nth place bit in the binary representation of the number and the number would reply with zero or one. It is all a question of where we locate the barrier between "double" and numbers.

In effect, the choice of location for expertise has been widened by regarding everything, including data, as process. Instead of only considering the two poles, intensionality and extensionality, we should utilize the whole dimension when deciding how much interacting processes should know about each other. We discussed similar problems of allocation of knowledge in connection with the distribution principle and clusters of functions in Chapter 1.

There is a further problem with the definition of intensionality and extensionality, which arises when we try to form a consistent idea of a sub-process. In Figure 8, the process F is able to find out about the structure of the compound process bounded by the dotted line. F can ask E and B who they are connected to and thus become acquainted with A. It is as though the large process were a society of processes and F were getting to know the society. In the realm of material objects, examples of directly examinable structures are easy to find. If I have access to the leg of a chair, I can locate the other parts, and if it is upholstered can find a spring by looking inside. In some cases, as in nuclear physics, direct examination is impossible and internal structure is only revealed by examination of behaviour and by model building. If intensionality is regarded as allowing behaviour, but not structure, to be examined then it becomes difficult to see how we can combine processes while preserving it.

## 3.4 Communication between processes

There are several points we should make about the type of communication which occurs between processes. First of all, communication is very important, so much so that Hewitt seems to make it the most important feature of a community of actors. A system's behaviour is composed of message-sending from one actor to another, and at the base of the system are primitive actors which have a given message-sending and receiving behaviour. We feel that "state" is as important as message-sending, even if all structures are considered as processes. We could build up a more complicated group of processes than that within the dotted lines of Figure 8 and then have another group get to know it. A process could send messages asking members of the first group what type they are, what state they are in and who they know, and by doing this could picture the state of the group.

It seems that the primitive for communication must be direct. A standard indirect way to allow two processes to communicate is via a buffer. Since we have decided to regard any data structure as a process we must think of a buffer in this way, so we have the question "How does either process communicate with the buffer?". To avoid the infinite regression, we think of communication as being direct and then, if we wish, we build clusters of processes which behave as whatever kind of interface we need.

We have considered two cases of function call between a function and its argument. In one case, the function operated on the argument - this was the extensional case - in the other, the function sent a message to the argument which itself was activated. If we consider that everything is a process we can extend this idea of communication. As mentioned in Chapter 2, we might want two processes to "react" when associated together. This type of "process call" might well be related to a conversation between two people. At first both people need to make introductory gestures and noises until they gradually get down to business. Then either may decide to get the other to do something for him. The storekeeper data-bases of Section 2.8.1.4 provide an example of this. A customer might go into an electrical shop and ask for a "13 amp". The proprietor would prompt with "13 amp fuse, plug or socket" and the customer would know

what was lacking in his specification without knowing the possibilities beforehand.

Just as the nature of the communication between processes need not be simple, so the messages themselves might have structure. We can envisage that as groups of processes become more sophisticated, the nature of their language becomes so. Then, with these groups as building blocks, we build bigger units with another step in complexity of communication. As the range of behaviours of groups of processes becomes large, it becomes necessary to introduce structure into the messages passed between them to prevent the need for a single symbol for each possible behaviour. This structure provides a language for describing desired behaviours for passing information and advice and for other purposes. Such languages are in existence already to allow users to operate programs such as editors. It would clearly be desirable to provide conventions so that entire groups of processes could communicate in the same way. This would allow modularity. As a first step, it would be an interesting project to design a simplified version of English and use it as a basic language between primitive processes.

Since the level of complexity of messages sent between processes can vary so widely it may be better to use two words to describe the exchange. If the message is a simple causal exchange then we might merely say the processes "interact", or that one process "acts upon" the other. We had an example of this at the beginning of the chapter. Pushing a physical object, such as a car, is an interaction rather than a communication. On the other hand, when the message is complex and is associated with intentions we could say that two processes communicate. Natural language is a good example of this. There is a range of possibilities between these two extremes. It can also be useful to think of the message as active. I might send a message into a building in the form of a messenger who will seek out the recipient, sum up the situation, and pass over an appropriate message. In human visual or linguistic perception it may be useful to think of the perceived actively reacting with the perceiver.

But at this stage it is premature to put forward such statements as anything more than possibilities. It seems that processes provide a

unifying element in computing which has not been examined deeply enough. A key seems to be the need for real parallism of processors. There is no reason why only one part of a memory should be active at once except that we provide only one processor. Research is needed to see what simple primitive processors could be used in parallel to allow any process to be implemented and to see how a "core" of these processors could communicate. Like the glider in the game of life, processes often seem to be an imposed property of some changing situation. We may well be on the wrong track if we try to bind a process to a fixed piece of text as do most programming languages. We must start with units of behaviour as our building blocks and find ways of putting these together.

# Chapter 4 GEOG-LINE

## 4.1 Introduction

In the last three chapters, we discussed knowledge representation and handling from a theoretical point of view to provide a basis for discussions of dialogue programs and their relationship to teaching. This chapter, together with the next two, describes some practical investigations into this theory. Each chapter discusses a program which exemplifies points in the theory and brings to light various problems and principles. Although our initial goal was to produce a working dialogue teaching program, the problems involved are so formidable and so interesting that we have instead addressed them individually with experimental programs. We only consider three of these in this thesis.

In this chapter, we describe a program named GEOG-LINE which takes part in a dialogue with a user in order to find out about regions which lie on a straight line world and have locations and properties. We use it to investigate problems of consistency of data, resolving contradictions, and structuring dialogue. At the end of the chapter, we present four principles related to this program. They are : the distribution principle, the dialogue principle, the postponement principle and the talk-to-yourself principle. The first of these is taken directly from Chapters 1 to 3 and the postponement principle and talk-to-yourself principles are related to it. Thus GEOG-LINE is strongly based on the theory put forward previously.

In Chapter 5, we will discuss CARTOGRAPHER, a program which takes part in a dialogue about two-dimensional maps. With CARTOGRAPHER, we tackle slightly different problems besides consolidating some of the results of this chapter. Chapter 5 considers rules of dialogue for asking relevant questions at appropriate times when solving a problem posed by the user, and considers the role of a procedural model of the user in this.

In Chapter 6, we will investigate ways in which explanations can be given to questions. We begin by investigating general situations and we classify "why" questions. Then we describe a program which gives explanations for certain simple cases. An important sub-principle of the distribution

principle, called the distribution of interpretation principle, is examined. This can be summarised as "each object of an intelligent system should have sufficient interpretative knowledge of its own to enable it to carry out the task it is designed for." We oppose this to the case where one global interpreter deals with all actions, and relate it to the theory of our first three chapters.

Before describing GEOG-LINE we first consider some properties of dialogue to set the scene.

## 4.2 What holds a dialogue together?

### 4.2.1 Simple Links

What then are the links which cement sentences in a dialogue? One way to link sentences is to use words specially designed for this purpose, i.e. the conjunctions "thus", "and", "but", "therefore", "though", "however", and so on. The sentences need not necessarily be uttered by the same speakers since we could have the exchange

A That is a car.
B Yes ! But it doesn't work.

Another simple way to make links is by using the various cross-referencing techniques available in natural language. For example, the same word used in several places, uses of "it", "the", "that" and other referencing words, and anaphoric references. Some words require various connections implicitly before they become meaningful. For example in

"The temperature is sixty degrees"

"temperature" refers to some place and some time which must be discovered for the sentence to take its full meaning.

For its own world, and in a certain number of cases, Winograd's program (Winograd, 1971) dealt with these simple referencings. Charniak

(Charniak, 1972) showed that finding a correct object for a reference might involve a great deal of reasoning based on real world knowledge. Consider for example

John gave Bill some books.
He wanted him to be happy.
John gave Bill some money.
He spent part of it on a present.

Charniak also showed that this reasoning allowed links to be made which are not directly signalled by some syntax like a special reference word. They are made simply by the juxtaposition of two sentences which have some semantic connection by way of reasoning.

John wanted some money.
He got his Piggy Bank.

This also occurs with our earlier case of conjunctions. They can often be missed out and the relation between the sentences will still be understood because of their semantic content. So, "because" is omitted in

I can't repair my roof.
There is a shortage of plasterboard.

### 4.2.2 Story Context

The larger subject under discussion gives any single utterance significance and makes it part of a structured whole. Charniak's thesis was based upon children's stories and by means of deductions, using real world knowledge, each sentence took its place in the structure of the theory as a whole. The same "topic" or "story context" can bind a dialogue together. Consider for example a dialogue between a parent and child where the parent reads a story and explains it while the child asks questions. Much of the continuity of this dialogue would be due to the structure of the story.

The blocks world of Winograd's program also exemplifies this kind of story

context. Continuity in this world helps to give continuity to the dialogue. It allows references back to actions and past states of the world, and in some cases there is a direct isomorphism between part of this world model and part of the dialogue. Consider for example such sequences as

Q Why did you pick up the red blocks?
A Because ...
Q Why did you do that?
A Because ...
Q Why did you do that?
A Because you told me to.

Here the goal structure of a past action in the model world maps on to the structure of the dialogue. This goal structure needs to be distinguished from the goal structure next discussed.

### 4.2.3 Goal Context

The particular conversational cement we will deal with concerns conversational goals or conversational purposes. An English utterance is generally spoken for some reason; it may be an order to effect some action, and the order might be responded to with 'OK' to express understanding and acceptance of the order. The purpose of a question could be to test someone else's knowledge or to increase one's own. Vows, insults and compliments have their own purposes, and so on.

There is a whole class of utterances called performative utterances, whose meanings are similar to their intention (Austin, 1962). Some such utterances are

I promise to give you sixpence.
I agree.
I take this woman as my lawful wedded wife.

Every utterance has intentions or purposes, although it may require subtle deduction (including contextual knowledge) to reveal these intentions.

People are amazingly efficient at discovering intentions. A very simple, often quoted, case where the intention is different from the literal meaning of an utterance is

"Isn't it cold in here?"

The intention may be to get someone to close the window or poke the fire, thus making what appeared to be a question into an order. Sometimes people miss the intention of an utterance even though they may be perfectly well aware of its literal meaning. So we hear statements like

"I don't see what you mean".

The intentions of utterances link together to form what we call a "goal context" in the same way as their contents form a story context. Utterances are put to many uses to accomplish the larger purposes of an entire dialogue. The goal context forms a context in parallel with the story context as the dialogue progresses, and both contexts may need the same mechanisms of reasoning, using any available information, for all their links to be discovered.

The difference between topic context and goal context is similar to the distinction made by Halliday (1970) between structure and function. This has been taken account of at the level of words and phrases - for example, a noun-phrase is recognised as being used either for subjects or objects - but until recently no major program we know of made explicit the functions of individual sentences within a dialogue. One recent exception is by Power (1974).

Most dialogue programs are really only question-answering or order-taking programs. In many ways they act like input-output boxes. A question is asked, the program is entered from the top level, and when the answer is found the program replies and exits. A session with such a program consists of a sequence of shallow goals, each to answer a particular question, and each separate from the rest. After each question is answered, the program has no goal context and is back at top level. The only differences in the program's state are changes in knowledge of the

topic, the conversational partner or the history of the conversation. There are no ongoing goals or representations of intentions.

We find that intentions and goals are so important to the structure of a dialogue that we state our requirement as a principle called the Dialogue Principle.

> "Utterances should be distributed throughout a program so that inputs and outputs are made within a context of goals and intentions".

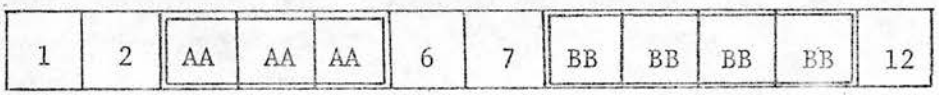
Despite their restrictions, hierarchies give a working approximation to many situations. We can consider the goal structure of a discourse to be a hierarchy. There are global and local goals, and main goals with sub-goals. The reason for an entire conversation might be to clear some matter up. It may have sub-goals to explain particular topics or to disagree with particular parts. In turn these may have sub-goals, altogether producing a hierarchy.

In our program, we link this hierarchy with the run-time goal hierarchy of a recursive programming language so that with particularly simple programming we can illustrate how utterances can be tied into a dialogue. Any programming language allowing sub-routines has a goal structure even if this is not explicit as in the more recent goal directed pattern matching languages. A function or procedure can be mentally associated with a goal which is effectively "to do what the function does". Then, sub-routines called by the function will execute sub-goals of the main one and the execution of the program can be thought of as producing a goal hierarchy. There is no reason why the goal hierarchy of a conversation should not be recursive. For example, a function to explain some topic given it as an argument, might split the topic into smaller portions and call itself recursively on each. We will see another example of recursive conversational goals in the program to follow.

## 4.3 Description of GEOG-LINE'S micro world and sample dialogues

### 4.3.1 The World

The world our program operates on is one-dimensional. It is a line of unit intervals numbered as in Figure 9.




AA is a REGION
XMIN = 3
XMAX = 5
SIZE = 3

BB is a REGION
XMIN = 8
XMAX = 11
SIZE = 4

Figure 9

GEOG-LINE allows objects with names. Regions are the only objects allowed on the world-line and a region may be a province or a country. Since a region has an extent on the line, it has three metric properties : the number corresponding to its left-hand end which we call its xmin, that corresponding to its right-hand end or its xmax, and its size, which turns out in this representation to be xmax-xmin + 1. The system is initialized to have no knowledge of the particular objects in its world, but it does know how to handle the types of objects and ask it about them.

### 4.3.2 Simple dialogues

Figure 10 shows a simple dialogue which we will discuss to give a preliminary view of the system before we explain some more sophisticated features. It is composed of five consecutive snatches of dialogue, indicated by the square brackets at the right-hand of the page. The only dialogue relations between the snatches are those relating to the changing world knowledge of the program. However, within each of them, the utterances are linked by goal sub-goal relationships. The program is entered at the beginning of each of the units and does not exit until the end. In the course of its execution the program may output to the user, and the user

```
   USER   :NEW-OBJECT A1;
          :A1 -> OF (EG, REGION);
PROGRAM   WHAT IS THE XMIN OF A1
      U   :2
      P   THANK YOU
          WHAT IS THE XMAX OF A1
      U   :6
      P   THANK YOU

      U   :NEW-OBJECT A2;
          :A2 -> OF (EG, PROVINCE);
      P   IS A2 A REGION
      U   :YES
      P   WHAT IS THE XMIN OF A2
      U   :DONTKNOW
      P   OH DEAR
          WHAT IS THE XMAX OF A2
      U   :11
      P   THANK YOU
          WHAT IS THE SIZE OF A2
      U   :5
      P   THANK YOU

      U   :NEW-OBJECT A3;
          :2 -> OF (SIZE, A3);
      P   IS A3 A REGION
      U   :YES
      P   WHAT IS THE XMIN OF A3
      U   :15
      P   THANK YOU

      U   :A2 -> OF (EG, COUNTRY);
      P   A2 IS A PROVINCE AND CANT BE A COUNTRY

      U   :A2 -> OF (PROVINCE, A3);
      P   IS A3 A COUNTRY
      U   :NO
      P   A3 IS NOT A COUNTRY SO IT HAS NO PROVINCE
```

Figure 10

may respond to the program.

The first line simply declares that A1 is the name of a new object we are introducing to the system. We say 'A1 is a region' (literally 'assign A1 to examples of regions', using the POP-2 updater facility - see Appendix 1, "Notes on POP-2"). This arouses the program's "curiosity" and it asks about the location of the region. To do this it has to execute three separate sub-goals, namely 'askabout xmin', 'askabout xmax' and 'askabout' size. For the first two of these it receives an answer and returns its thanks, but the last goal does not generate a question since the program can evaluate the size itself from xmax and xmin. It is satisfied and exits. A small point, but worth noting, is that one of the program's utterances is made up of two separate parts belonging to different goals. This is "thank you, what is the xmax of A1".

There are two differences in the next group of utterances. Firstly, we say that A2 is a province. Now a precondition for this is that A2 is a region. The program asks itself this and discovers it does not know. So it asks us instead. On discovering that A2 is a region after all, it calls the same routine as before to ask about it. In this case, however, it does not know the xmin of A2. So after asking for the xmax, the program also asks for the size.

In the third example we say "the size of A3 is 2". Again there is a precondition since an object which has a size must be a region. The program again asks and is answered by 'yes'. This time all the "askabout" routine needs to ask about is the xmin. This is because the purpose of the main goal is to assert the size of A3. Even if we answered 'don't know' to both a request for xmin and xmax, the program would refrain from asking us the size of A3.

The last two example illustrate two more preconditions, namely that a province cannot be a country and that anything which has a province must be a country.

There are various functions which allow the program to communicate with the

user. First of all there are about twenty functions which print messages as templates with certain variable parts that are given to the functions as arguments. These functions are structured by a method involving POP-2 partial application (explained in Appendix 1) so that besides being executable they can also be examined. The message "place1 is not a country so it has no provinces" would be

```
because(%isnot(%place1,country%), hasno(%province%)%)
```

which is a tree structure. The point of the tree structure is that it is built from procedures like "because", and "place1" which when given argument tree structures and asked to print themselves out know how to ask their arguments to print out and how to print out connecting phrases for their arguments. In GEOG-LINE, these messages are not examined by the program itself as communication between functions is simple. All the users' inputs, apart from the initial one, are special format answers to a program question. Ideally we should allow the user to say anything admissible at any time.

Secondly, there are the more generally used functions, GET, ASK-IF, ASK-IF-RIGHT, and CHECK-AND-ASK, which embody simple rules about asking questions. We will see instances of more complex rules for this in the next chapter.

(a) GET is a function which tries to find out from the user the value of a relation. Its result is either true or false, and if true it also produces the value. It sends appropriate responses to the user in either case.

(b) ASK-IF is used when the program thinks something ought to be true. It asks the user and accepts the responses "YES", "NO" and "Q". "Q" for query is interpreted as "YES", and sends a message to the user to inform him of this.

(c) ASK-IF-RIGHT is used when the program used to think something was true but is now uncertain of it. If the user says "YES" or "Q", the program does nothing; but if the user says "NO" the program erases its incorrect knowledge. This function is

used in resolving contradictions.

(d) CHECK-AND-ASK is used when the program wants to know if something is true before it has checked to see if it knows already. The program first asks itself. If it doesn't find out then it asks itself the opposite. Otherwise it asks the user. If the user says "YES" then the program attempts to assert the fact. If this is unsuccessful then a failure message of some sort is printed out to the user.

These four functions represent some simple rules of question-asking. It would be interesting to have parts of the program communicate with other parts according to similar rules. This would be useful in designing a larger system which decomposed naturally into several components. To some extent this has been achieved by Power (1974) who modelled two cooperating processes which use language games to solve a common problem.

### 4.3.3 Dialogues involving contradictions

So far, we have two reasons why the program might decide to ask the user a question or tell him something. They are

(a) The "curiosity" of the program which causes it to ask for information about new objects.

(b) Checks for preconditions which may excite a query, or a statement that the precondition doesn't hold-

We now introduce further conversational goals relating to contradictions between the system's knowledge and the user's. These are

(c) To detect contradictions.

(d) To alert the user to the contradiction.

(e) To attempt to resolve the contradiction by dialogue with the user.

As the program stands it cannot tolerate contradictory information and will not rest until it is satisfied. The contradictions we have treated are concerned with the breaking of two constraints

(1) The XMIN of a region is less than or equal to the XMAX.

(2) The SIZE is compatible with the XMIN and XMAX.

Figures 11, 12, and 13 show example dialogues involving these new rules. The system also notices whether XMIN, XMAX, and SIZE are consistent with what it already knows.

```
   USER   :NEW-OBJECT A3;
          :3  OF(SIZE,A3);
PROGRAM   IS A3 A REGION
      U   :YES
      P   WHAT IS THE XMIN OF A3
      U   :Q
      P   OH DEAR
          WHAT IS THE XMAX OF A3
      U   :Q
      P   OH DEAR
          OK
      U   :4->OF(SIZE,A3)
      P   THAT CONTRADICTS WHAT I KNOW
          THE SIZE OF A3 IS 3 SO IT CAN'T BE 4
          I THINK SIZE = 3 IS THAT RIGHT
      U   :NO
      P   OK
```

Figure 11

In Figure 11, we first set up a situation where all the system knows about A3 is its size, which it believes to be 3. We then say "the size of A3 is 4" which contradicts this. The system points out the contradiction and tries to rectify it. It then asks the user if its conclusion is correct.

```
    USER    :NEW-OBJECT A2;
            :3 -> OF(SIZE,A2);
 PROGRAM    IS A2 A REGION
       U    :YES
       P    WHAT IS THE XMIN OF A2
       U    :5
       P    THANK YOU
            OK
       U    :4 -> OF(SIZE,A2);
       P    THAT CONTRADICTS WHAT I KNOW
            THE SIZE OF A2 IS 3 SO IT CAN'T BE 4
            I THINK XMIN OF A2 IS 5 IS THAT RIGHT Q
       U    :NO
       P    OK
```

Figure 12

In Figure 12 we tell the system the size and xmin of A2. In fact, as will be discussed later, the system 'prefers' to know the xmin and xmax rather than one of them and the size, so its set of facts about A2 is now xmin = 5; xmax = 7. It can easily work out the size from these figures, so when we state that the size is actually 4 we again have a contradiction similar to that in Figure 11. However, there is a difference. Since the exact location of A2 is known, the contradiction now implies that either or both of the endpoints of A2 are wrongly located. In fact, our program only allows one to be wrong and asks, in turn, whether it is the xmin or the xmax. In the example, we tell the program that its xmin of A2 is wrong. So the program sets everything as it should be assuming the size is 4 and exits. If at this point we had answered 'yes' instead of 'no' we would have been asked about the correctness of xmax. If we had again said 'yes' the program would have no alternative but to assume it was in fact correct about the size, and it would respond "then the size can't be 4". The difference between the examples of Figure 11 and Figure 12 is

that in the first case we have a straight contradiction of a fact, and in the second we have a contradiction of a result deduced from several facts, each of which may be incorrect.

```
    USER    :NEW-OBJECT A1;
            :2 -> OF(XMIN,A1);
 PROGRAM    IS A1 A REGION
       U    :YES
       P    WHAT IS THE XMAX OF A1;
       U    :1
       P    THANK YOU
            SORRY THAT'S A CONTRADICTION
            THE XMAX AND XMIN OF A1 ARE INCOMPATIBLE
            I THINK XMIN OF A1 IS 2 IS THAT RIGHT Q
       U    :YES
       P    WHAT IS THE XMAX OF A1
       U    :1
       P    THANK YOU
            SORRY THAT'S A CONTRADICTION
            THE XMAX AND XMIN OF A1 ARE INCOMPATIBLE
            I THINK XMIN OF A1 IS 2 IS THAT RIGHT Q
       U    :NO
       P    WHAT IS THE XMIN OF A1
       U    :Q
       P    THANK YOU
            OK
```

Figure 13

In Figure 13 the contradiction occurs while the program is asking us about a new region. We say that the xmin of A1 is 2 so the askabout routine asks us for the xmax. However, the xmax we give is less than the xmin so the program attempts to resolve the contradiction by entering a function designed for this. It checks to see who is right by asking the user. As with the last example, this incompatibility could have two causes : either the xmin or the xmax may be the culprit. Xmin turns out to be alright, so xmax must be wrong and the program asks us for the correct value. We again give an incompatible answer so while still in the process of resolving the first contradiction the program enters the resolving program again. This is an example of the dialogue becoming recursive. Since knowledge about resolving contradictions has been put in a sub-routine we can use it anywhere. In other words, whenever we assert any new information, even that found out during the discussion of

a contradiction, we apply a standard contradiction checking and correcting routine to it.

The run-time structure representing the stack of functions entered at this point is shown roughly in Figure 14. We could utilize this run-time context by making the print-out context-sensitive so that at the second contradiction the program says "Sorry, that's a contradiction too". We could also use the variables attached to it as a history of the discussion to put the user right if he contradicts himself or is otherwise confused. As the program is currently written it always acts as if there is only one contradiction which is the most recent one.

```
top level
declare xmin = 2
askabout the new region
ask  for xmax and then declare it
resolve the contradiction
ask for xmax and then declare it
resolve the contradiction
```

Figure 14

The run-time structure shown in Figure 14 represents what the program is doing and is produced naturally by executing a program consisting of various goal satisfying functions, each of which may call several of the others. The method is effective at producing dialogues in this simple situation mainly because knowledge about dialogue has been represented in procedural form and therefore in an ideal way for causing a dialogue to happen. It is interesting to consider how much extra information has to be added to the bare facts about the location of regions to bring these facts to life and make them of use during the execution of a program. Whether the same method will work on larger systems is an open question.

Our principle of distributing utterances throughout a program so that inputs and outputs are made within a context of goals and intentions represented by the run-time structure, was most useful in structuring simple dialogues. With larger systems, different and more expressive types

of run-time structure may be needed or, more likely, intentions and goals may be made explicit examinable structures so that reasoning can be done about them.

It may be fruitful to design a program using process-oriented structures rather than function-oriented ones. Let us take a simple example of how this idea might be used to explain to the user about some topics in physics. This requires a preceding goal to be executed to make sure the user knows enough mathematics by teaching him some. Now, while executing the 'teach physics' goal it might discover that still too little mathematics had been taught, requiring a continuation of the 'teach mathematics' goal for a while. The precise restart point for 'teach mathematics' might depend on context, in particular on the nature of the gap in knowledge. Thus the 'teach mathematics' process must be versatile enough to accept advice from the 'teach physics' process and to react appropriately to its run-time environment. It is like a mathematics teacher who can remember about his pupil and what stage he had reached in his tuition but can still adapt to particular needs at any time. This supports the ideas about versatile bundles of skills put forward in Chapters 1 through 3.

## 4.4 Principles

In any discussion of an AI program, we can raise the question "But you told the computer how to do its job - it didn't do anything new on its own". There is a whole dimension at one end of which a program will slavishly follow instructions in a pre-set order and at the other will use more of its own initiative. It is important when we think about programs which are modelling or illustrating situations, especially when we anthropomorphise by describing them in terms of "wants", "decides", "prefers", that we see exactly what it is that the program does, and where dividing lines occur between the program's decisions and those of the programmer. From this present program, we will take some examples of knowledge which is programmed in a very pre-set way, and will describe how it could be made more flexible and its deployment more easily attributable to the program's own initiative. An important principle concerning generalisation of programs emerges from these examples.

Firstly, consider our statement that the program "resolves contradictions". It certainly is true, but to what extent, and how does the program do it? Which contradictions does it resolve, for example? We find that it only resolves implied contradictions concerned with the relations between xmin, xmax, and size, and direct contradictions of data such as xmin, xmax and size. There were so few that we could program them specifically and separately. We knew, as the programmer, that at certain points in the code certain contradictions could arise and at these points we inserted a code to deal with each particular one. So we have made decisions for the program already. We decided where contradictions will occur and which information the new information contradicts. The first of these is difficult for the program to do by itself. The knowledge that "contradictions may occur when new information is received" is best given to the system by means of a demon which checks all the time to see if new information is being received so that it can ask "could this information contradict anything I know?". The second requires the program to find an answer to this question. To do this the program must be able to tell what sort of information might be contradicted by new information. As programmers, we made a distinction between direct contradictions of new information with old information, and contradictions of new information with deductions made from old information. This is one thing that the program ought to do. It requires the program to answer the question "What constraints does this new information take part in?".

Secondly, we said that the program "exhibits curiosity", in that information about a new region will cause it to ask the user questions about that region. This could be treated in a way similar to the first example. We would require a demon which was watching for new information, and which would ask the question "What might I want to know about this?". The program would need some mechanism for answering the demons question and in return for each answer would ask the user a question.

Thirdly, we described the program as "preferring some representations to others". It stores the xmin and xmax of a region rather than the size since direct information about location is more useful than size. This

was a programmer's decision. More generally the program should ask itself each time new information is received "Is there a more useful representation of this knowledge?". This would need sophisticated sub-routines. If a program does change the representation of the information it is given it needs to keep an audit trail of how it came by its information, or why it believes it, so that it is able to deal properly with contradictions when they arise.

Let us try and abstract some principles from these three examples by seeing what is common about them. In each case there were certain points in the program identifiable by contextual happenings such as "new information has just arrived". At each of these points we had special pieces of code, all of them differing. We replaced these pieces of code by one question which was the same in each case. This was a question the programmer had asked himself when he chose the special code. In effect, the special code is the particular answer to a general question. This process simplifies the programs which use the similar pieces of code by replacing them with one sub-routine. The principle is

> Make implicit knowledge into explicit knowledge by replacing many special purpose pieces of code with code that asks a general question and by giving the system the power to answer that question in many contexts.

This technique leads to a postponement of action. Instead of doing a special purpose function the program asks another function to tell it how to act in the circumstances. This function again might ask further functions. The program's behaviour is more implicitly related to its code, whereas many individual steps are explicitly taken rather than being done by the programmer. In plain words the more adaptive a program is, the more it must "know" what it is doing.

The technique just described also illustrates a principle we call "the program talks to itself" since we insert lines of code which correspond to questions. Notice from the examples that each initial question or statement given to the program is actually written in the programming language, POP-2. Because of this, the program can use just this form to talk to itself. If we say that "the size of A1 is 4", the program will ask itself "is A1 a region" in just the same way we would have asked it.

We could even interpret any subroutine call as the asking of a question. Calling a subroutine to find the square of three is equivalent to the question "what is the square of 3?" or the order "evaluate the square of 3". Pursuing the idea, we can see that a program which carries on a dialogue intelligently with a user should consist of modules which carry on intelligent dialogues amongst themselves. This is illustrated in Figure 15.

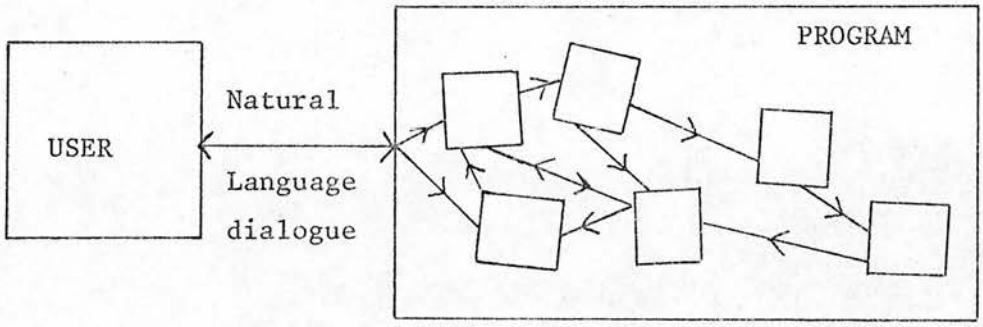



Figure 15

In this light we see a big discrepancy between the quality of communication aimed for between a user and a program as a whole, and the quality usually allowed between the component functions of the program. While this argument is only by analogy, it strongly suggests that we must narrow the gap between the two levels of communication. We can run the argument backwards. Suppose we restricted communication to simple argument passing and did not allow dialogues between the components of a system. If it would still be possible to write a program capable of natural language dialogue, why could not the components of the program communicate in the same language as the program as a whole? This approaches arguments relating language and intelligent thought. It suggests that mechanisms for thought could be expressed as programs constructed of instructions phrased in a powerful natural style language. An immediate implication of this is that components of such a system should themselves behave intelligently, taking note of the environment in which they are executed and reasoning instead of following pre-set paths of instructions. In particular, one qualitative difference between the two standards of communication is that our program as

a whole is supposed to have a dialogue, whereas functional programming is simply a case of one input and one output. What are needed are processes with which we can communicate by dialogue and which remember about earlier calls of themselves for use in later ones. We need functions which can be exited and later re-entered. P-74 (see Part II) is the kind of language which will allow a process to be structured into sub-units of processes entering into dialogues between themselves, with each sub-unit itself structured in a similar way.

The requirement that a program should be made of self-sufficient sub-units which can adapt to varying situations is very reminiscent of the distribution principle of Chapters 1 to 3. We can illustrate this from GEOG-LINE by considering a routine called "ASK-ABOUT" which has some self-sufficiency due to which the programming of GEOG-LINE was simplified. The purpose of ASK-ABOUT is to ask for values of xmin, xmax, and size. We have seen that these requests are not independent. It would have been possible to express the dependence by giving ASK-ABOUT a fairly complex flowchart, including statements like "if the answer to xmin was 'dontknow' then ask for xmax else ask for size". The more properties we have to ask about then the worse the flowchart becomes. It is possible, however, to give ASK-ABOUT the very simple structure shown in Figure 16.

```
TO ASK-ABOUT X;
ASK-FOR (XMIN,X);
ASK-FOR (XMAX,X);
ASK-FOR (SIZE,X)
END;
```

Figure 16

The price we have to pay for doing this is to make ASK-FOR independent of the environment in which it is called. It must make two checks. Firstly, it must ask the program "do you know xmin?", say. If it does know, ASK-FOR exits. If not, it must ask the higher goals "am I already being asked for the xmin or being told the xmin?". If the answer to this is 'yes', it also exits. Otherwise it asks the user for xmin and stores its result. Sometimes,

therefore, ASK-FOR (size) is entered when xmin and xmax are already known. This unproductive entry and exit may seem inefficient but it makes the program much easier to handle.

Combining the principles we have just discussed, we can make the following useful summary.

1. The dialogue principle

   Utterances should be distributed throughout a program so that inputs and outputs are made within a context of goals and intentions represented either by the run-time structure or other manipulable structures. This principle opposes itself to programs where utterances are always associated with top-level goals which invoke procedures that do not interact with the user.

2. The distribution principle

   Intelligent, adaptive systems should be modularly constructed from intelligent self-sufficient subsystems which have their own procedural knowledge about how to do their repertoire of tasks.

3. The talk-to-yourself principle

   The subsystems of an intelligent system should communicate with each other by means of dialogue in a rich language. This is opposed to the single passing of simple parameters between subroutines.

4. The postponement of action principle

   Many actions should be postponed by preceding them with explicit decisions about which of several actions should be performed. This leads to self-sufficient programs which make decisions in context and thereby follows the distribution principle. Also, much implicit procedural information embedded in a program by

its programmer is then made explicit by generalisation of many special purpose pieces of code into a single subroutine representing a question answered by the programmer during programming.

## 4.5 Extensions to GEOG-LINE

Some extensions to the system would merely increase its size without making any qualitative change. Examples of these, are : to include towns; to allow regions to be land or sea; to make it a constraint that a province of a country must be within that country; and to take account of relationships such as an island is a piece of land with sea on both sides, sea may not have towns on it, a town cannot be on the sea.

The only difficulties in making all these additions are in keeping the data consistent. GEOG-LINE detects contradictions immediately, if at all, and then rectifies them. Its goal is absolute consistency within its powers of detecting contradictions. We will describe one extension we have made to GEOG-LINE which shows how the goal of maintaining absolute consistency is misdirected and sheds some light on how people might assimilate information. The next chapter will discuss a program called CARTOGRAPHER and adds more weight to the argument.

For our extension to GEOG-LINE we added the constraint that regions should fit onto the world line in a consistent way. They should form a partition of the world line with no regions overlapping and the entire world filled up. We considered the world to be of a fixed given length and only allowed one kind of region, to avoid the problem that provinces also form partitions of the countries they are in.

Detecting inconsistencies is now a tricky problem. If we knew exactly the position of every region we were told about, we would have no trouble; but when we have some regions whose size alone we know, the problem is more complicated. We have some regions whose position we know. This leaves holes in the world line. There are some regions for which we can only locate one end. So we must assume they are the shortest possible length, i.e. 1 unit. Then there are regions about which we only know the size.

We must check to see if these will fit into the holes left on the world, a problem which is either combinatorial or heuristic and which is far from general.

Having detected the inconsistency we have still more problems in trying to correct it. Again, if we only dealt with exact locations of regions we would have no trouble. There would be few regions which could be the culprit, namely those which the new region overlapped, and we could quite easily write a correction routine for this problem. But if we also allow unlocated regions with given size we have a much more difficult situation to deal with completely. There are very many ways in which the contradiction could be resolved. In Figure 17, for example,

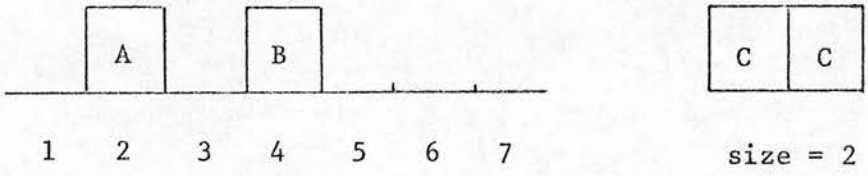



Figure 17

a new region of length 3 could be added if A was moved to the left by one unit since C would then fit between A and B, leaving room for D in the space of size 3. A complete method for finding the error would produce boring dialogue. A more human-like routine would be incomplete and might involve heuristics specifying sorts of things which might be altered to fit the new region and knowledge about types of mistakes that may have been made by the user. Methods for dealing with a contradiction might also depend on the past history of the conversation. For example, if we give the xmin of a region and its size, the program stores the xmin and xmax since it prefers this representation. However, in case of later contradiction it would be better to question the xmin and size rather than xmin and xmax.

In short, the problems of structuring the dialogue have now become very complex and are largely dictated by the structure of the situation being talked about rather than by complexity of rules of dialogue. If we allow

both provinces and countries, and include constraints of position due to them, the situation becomes even more complicated. In situations like this where people discuss complicated topics, certain domain dependent strategies for conveying the information become accepted. In our case, for example, if a person were to give the program data about most of the countries and provinces in the world he would perhaps locate the countries and, after each one, locate all the provinces of it. Then problems of contradictions and of relating incoming information become easier. According to our Dialogue Principle we express this as the introduction of larger goals such as "to give all the data about a province".

GEOG-LINE has provided a basis for some principles concerning dialogue and knowledge use and assimilation. In the next chapter we consider CARTOGRAPHER, a program which strengthens the results of this chapter and builds on them to develop more ideas about dialogue and to introduce the idea of models of the uses in dialogues about simple facts.

# Chapter 5 CARTOGRAPHER : A program to investigate data-base management and models of a user's knowledge

## 5.1 Introduction

In this chapter, as in Chapter 4, we study a program which maintains a data-base of simple facts by conducting a dialogue with the user. We call the program CARTOGRAPHER since it deals with properties of a simple map of the world.

Some of our concerns with the program GEOG-LINE were firstly how the program might detect and rectify contradictions in its data so as to keep a consistent data-base; secondly, how the program might ask relevant questions to fill in information it regards as important additions to its data; and thirdly, how to maintain a dialogue with the user to satisfy the first two aims. Here, again, we consider how a knowledgeable program can cope with problems of consistency and non-redundancy of its data, and we give convincing evidence against attempts to achieve completeness for either of these properties. The system we will discuss here does not attempt to resolve contradictions. Instead, we make it more able to detect them but cause it to refuse any contradictory information. This is the first part of the chapter. We next give some rules of dialogue which allow our program to ask relevant questions of the user when trying to answer a question posed by him. The rules are quite natural rules, so it is surprising that after serving well for many situations they break down and cause a reappraisal of the entire approach resulting in a statement of an improved set of rules which we have not implemented. The original dialogue rules forced us to consider procedural models of the user's knowledge and this is the third topic we discuss. It leads naturally into a more general discussion of pupil models in teaching situations with which we conclude the chapter.

CARTOGRAPHER stores information about a greatly simplified map of the world. For every location on this map it knows whether that location is sea or land and mountain or plain, and it assumes that the user does not have this information available to him. On the other hand, the user knows the locations of named places and perhaps knows properties of these places or relationships

between them. The program knows nothing about these named places and can only find out about them and locate them on the map by accepting information from the user.

The fact that the user and the program have complementary information provides a reason for the dialogue. The user informs the program and also queries it; the program answers the queries if possible and asks the user for information which may help it to do this.

Because we needed both a data-base of primitive relations and also a set of procedures to do deductions on this data-base, we used the programming language POPLER (Davies, 1971), an Edinburgh equivalent of MICRO-PLANNER (McDermott & Sussman, 1970). This language provided us with goal-directedness, pattern matching, the possibility of alternative methods to achieve any goal, and demons which are procedures that are invoked upon alterations to a global data-base. To show that complete non-redundancy and consistency are impossible to achieve we tried to bring the program as close as possible to achieving them so that the snags became evident. The backtracking regime under which POPLER runs lends itself to this approach but results in extremely inefficient programs. POPLER provides no help at all in designing a program which deals with our problem domain efficiently.

## 5.2 Consistency, non-redundancy and circularity : three problems of data-storage

The problems of representing cartographical information in an associative data-base are most interesting and shed light on representation in general. The two aims of a consistent data-base and a non-redundant representation are too rigid. A complex system needs to be able to live with contradictions and to be able to handle them as it discovers them, rather than attempting to trap them as they first become possible. Similarly, it is sometimes necessary to have redundant representations to cut down what would be very long computing times. If redundant information is to be removed, it is often much more efficient to have occasional reorganisation rather than a redundancy check whenever new information appears. These problems rise in representing knowledge of the predicates and relations

| | | |
|---|---|---|
| LATITUDE | LONGITUDE | NORTH |
| SOUTH | EAST | WEST |
| MOUNTAIN | PLAIN | LAND |
| SEA | COAST | NEIGHBOUR |
| NOT | | |

Figures 18 and 19 show the two maps which we provide for the system. Figure 18 is a map of land masses and Figure 19 is a map of distribution of mountains. Entries in the maps are either 1 or 0, indicating the presence or absence of the feature concerned. This means that mountains are either up or down - when they are only half way up they are either up or down. The scale of the two maps will indicate the crudity of the model.

Map of Land Masses

Figure 18

Location of Mountains

Although the model has complete world maps, it also needs to store incomplete maps in the form of assertions about places. This is necessary because the user may supply information about places whose locations are not yet exactly known by the program. As this information becomes more complete the program itself may be able to work out exact locations and assimilate the information into its standard representation.

The interactions between the relations listed above are surprisingly complex. It only seemed possible to deal with them by imposing some mental organisation on them. We tried to think of the entire data-base as a society of smaller data-bases which communicated, although we did not actually implement the system in this way. It would be an interesting project to find a clean way of expressing this modularity as it follows the principles of Chapters 1 to 3.

One module concerns the relations latitude, north and south. It is very similar to the corresponding longitude module, though not identical, and it is composed of five sub-modules. Let us first consider these in turn and then consider their interactions. Schematically the module is shown in Figure 20. The user communicates with the module by communicating with any of the sub-modules.

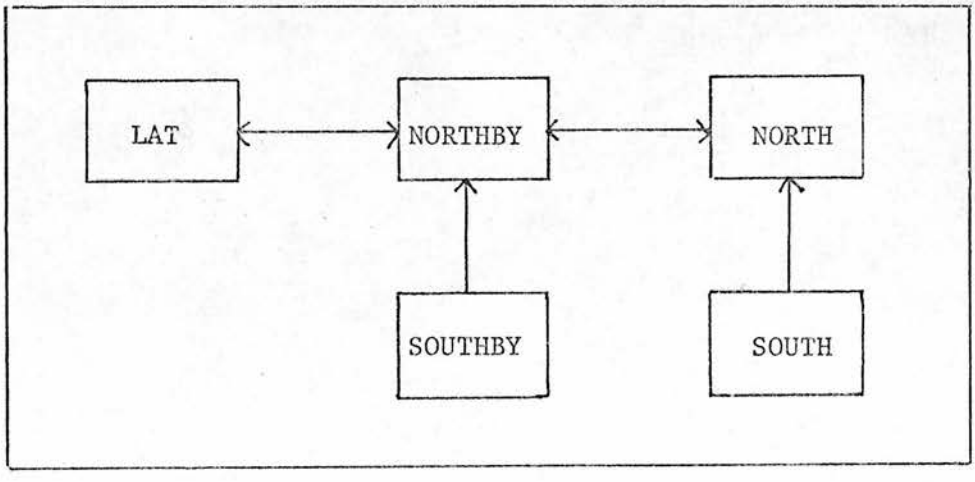



Figure 20

When we make an assertion about the latitude of any place to the module LAT, the assertion is stored and any question about the latitude of a place

invokes a simple retrieval. We can say, for instance, ASSERT <<LAT P1 30>> and can then ask GOAL <<LAT P1 £>X>>.

Assertions in the NORTHBY module are in the form

<<NORTH P1 P2 30>>

which means P1 is north of P2 by 30 degrees. Now, any set of assertions like this should form into connected subsets, each subset representing a line of places with known distances between adjacent places on the line. We arranged to have demons watching over assertions of the above form to see if the places mentioned could be slotted into any known line of places. A demon is a procedure which waits for certain changes in the data-base and is then invoked.

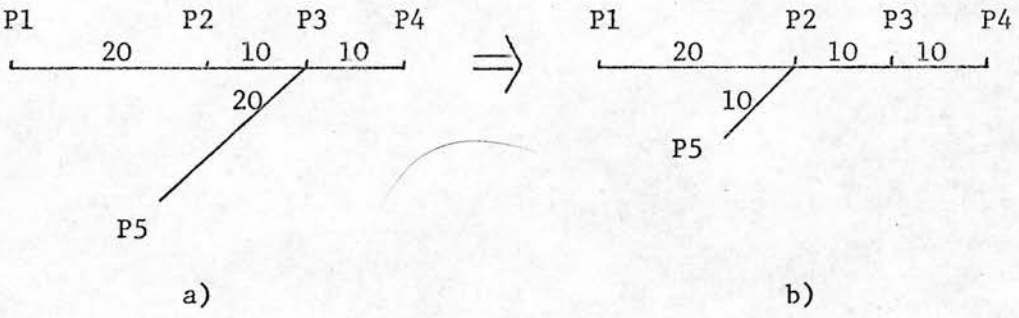
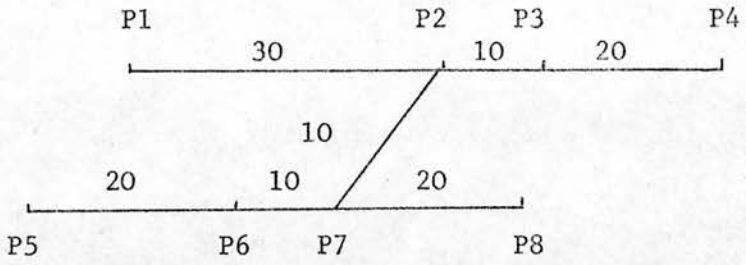



Figure 21

Figure 21 shows how these demons work. We assert that P3 is north of P5 by 20 degrees. The demon is invoked and searches for other places that P3 is north of by a known distance. It finds P2. The demon sees that P2 is nearer P3 than is P5 so it alters the data-base to the situation in Figure 21b) where "P2 is north of P5 by 10" is being asserted. The demon is called again. This time it will insert P5 between P1 and P2. This demon effectively eats its way along a line in one direction. We have a similar demon to take care of the other direction and they co-ordinate beautifully to deal with the situation in Figure 22.




assert <<north P2 P7 10>>

Figure 22

Assertions in the NORTH module behave in a similar way to those in NORTHBY. Since assertions are of the form <<NORTH P1 P2>>, where no distance is specified, at any time the data-base holds a partially ordered set of places. Again, we have demons to maintain a concise representation of this partial ordering. Figure 23 shows a demon transformation. The demon looks for triangles involving the new assertion and replaces them by lines. Depending on the side of the triangle involved, there are three possible demons. One of them effectively says "if I already know what I'm asserting then don't bother" and the other two say "if there is anything I have asserted which I can now work out, then erase it".

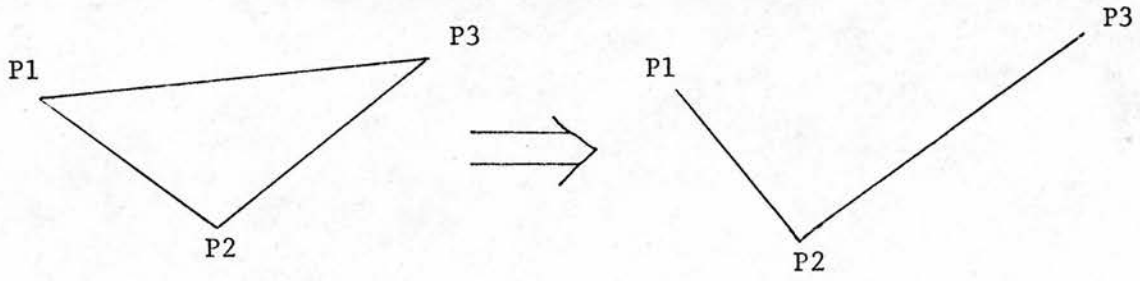



Figure 23

Let us now deal with the interactions between the modules. SOUTH and SOUTHBY interact with NORTH and NORTHBY by inverting their arguments. If I say "is P1 south of P2" this becomes a request "is P2 north of P1" and so on.

Whenever a new latitude is asserted LAT looks at NORTHBY to see if there are any more latitudes which can now be worked out. Similarly, any new NORTHBY assertion might refer to a place whose latitude is known. This would invoke a demon to walk down the relevant line of places asserting LAT's and erasing NORTHBY's. NORTHBY informs NORTH of any new NORTHBY relations that are asserted since NORTH need not keep a duplicate copy of the information.

Similarly, the three sub-modules depend upon each other for inference making. If I ask if P1 is north of P2, my request goes to NORTH. If it fails it will ask NORTHBY and if NORTHBY fails, NORTHBY will ask LAT.

Let us return to see how this particular module sheds light on general problems of data-base management. Firstly, consider the problem of preserving consistency of the data-base. If we tried to add the condition that latitudes fall within the range -90 to +90, we would run into trouble. Suppose for example we told that P1 was 30 degrees north of P2 which we knew was 160 degrees north of P3. This is impossible. Again, we might know that P2 was at latitude $80^{o}$ which would put P1 in an impossible position. As we extend our data-base, the difficulties of checking for inconsistency become arbitrarily bad. For instance, I might relate the positions of several places and state some of their properties as in the sequence of assertions.

```
<<MOUNTAIN P1>>
<<LAND P2>>
<<LAT P1 20>>
<<LAT P2 20>>
<<WEST P1 P2 10>>
```

The last assertion produces a contradiction since a search of the world map at latitude 20 degrees would show there exists no such structure. In all cases, the natural solution seems to be to take care of the contradiction when it arises.

Secondly, consider the problem of removing redundancy. If we make NORTHBY remove redundant assertions from NORTH we find that the simple triangle demons of NORTH won't work. One side of the triangle might be inferable from a chain of NORTHBY assertions. The amount of work to be done to find

all cases of redundancy increases dramatically and it becomes more efficient to spring-clean the data-base every so often instead of attempting to maintain absolute cleanliness.

A related problem to this occurs when we have assertions about two places which are in fact the same. If we say P1 is 10 degrees due north of P2 which is 10 degrees due east of P3 and P1 is 10 degrees due east of P4 which is 10 degrees due north of P5, the system should notice that P3 and P5 are identical and should try to merge them. There are problems both in noticing and in merging.

We can combine these observations about data-base self-management into one generalisation concerned with the serach for completeness. Both problems arose from the misdirected attempt to find all ways for doing something. In the first case, it was for all ways of detecting inconsistency, and in the second, it was for all ways of detecting redundancy. It is more likely that intelligent systems have only some ways. Their important feature is that the particular ways fit together in a sociable and appropriate way.

There is another problem we came across which arose for a similar reason, namely that the different ways of solving any subgoal did not fit together sociably. In POPLER a subgoal is represented by a pattern and the methods which could achieve the subgoal are procedures which have associated patterns that match the subgoal pattern. Several methods may match any subgoal and they are tried until one succeeds. Hewitt showed that this allowed procedural embedding of particular uses of rules so for example, the rule A implies B can be associated with either a goal to prove B or a goal to prove not A. The freedom to specify which uses of such a rule are to be allowed gives a very much improved efficiency in many types of problems. In our domain, this was not always so and there were embarrassing counterexamples which created our problem.

Consider the rule "A place P1 is north of P2 by n degrees if its latitude is greater than the latitude of P2 by n degrees". We could utilize this rule in various ways including

1. To solve the goal <<NORTH P1 P2 £>N>>, a procedure would be called to find the latitudes of P1 and P2.

2. To solve the goal <<LAT P1 £>x>>, a procedure would be called to find places north or south of P1 by a known distance and with known latitudes.

Both of these ways seem reasonable but if we allow both we have a problem of circularity for each is a subgoal of the other. 1) would call 2) which would call 1) and so on. In general, when we have many goals with many associated methods, the problem of detecting loops becomes more and more difficult as the loops are implicit.

There only seem to be two ways of controlling this situation. Firstly, we could try to detect loops as they occur by examining the current subgoal tree whenever a new subgoal is attempted. If the goal already occurs higher up the tree then we are in a loop. This check seems very inefficient. We used another method in our system. It is not entirely satisfactory but is workable. Our method was to prevent loops ever arising by putting data and methods in a hierarchy where we could be certain that control would always percolate downwards through the hierarchy. So, in the latitude, longitude north module which we described earlier in the section, "latitude" was lower than "north by" which was lower than "north". To work out how far P1 was north of P2, we could use information about latitudes; however, to work out the latitude of P1, we could not use information about how far north P1 was from P2. In other words, rule number 1) given above was allowed but rule number 2) was not.

This meant that another way of representing rule 2) was needed. We used the idea of a canonical representation for our ground facts. If<<NORTH P1 P2 N>> was asserted and if it was possible to evaluate any latitudes at that time, these latitudes were evaluated and stored. If this allowed<<NORTH P1 P2 N>> to be deduced we also erased this but this is optional. Rule 2) is now never needed. Instead, it is replaced by an antecedant procedure which puts the data in a canonical form.

To sum up, we have used antecedant and consequent procedures in a way which

places a hierarchy upon goals and so prevents loops. The reason this method is not entirely satisfactory is that a great deal of computation may be needed whenever a new fact is asserted. It may be better to leave the fact as it is and only do the computation when necessary.

There is one particular kind of loop which deserves special consideration since it is involved with the representation of negative information. We will put the view that negative information is always assimilated by means of antecedent procedures and is never stored. Either its implications give positive information or else it is forgotten. The loop arises quite simply. Suppose we try to prove the goal <<LAND P2>>. One of many methods for this is to try to prove <<NOT SEA P1>> which has a subgoal <<SEA P1>> which must not be achievable. To test that <<SEA P1>>is not possible we could try the equivalent goal <<NOT LAND P1>> and here we are almost around in a loop.

We only allowed the statements <<LAND P1>> or <<SEA P1>> to be present in our data-base. A negative statement like <<NOT SEA P1>> would be converted into <<LAND P1>> . No method for achieving a goal was allowed to generate a subgoal which was explicitly of a negative form. We tried to show that it was not possible to prove a goal, rather than it was possible to prove the opposite of the goal. Thus the results of a goal statement were either "The goal is proved" or "The goal remains unproved". This was one of the distinctions made in the "Farmer Brown" program of Davies (1974).

Only at the top level, when a user asked a question of the program, was a negative goal set up. This is discussed in the next section when we discuss dialogue rules.

It seems reasonable that negative information is not often directly stored. In the case of a binary choice such as LAND/SEA it is very easy to convert from the negative form to the positive form and both poles of the binary choice may be considered equally positive. In non-binary cases such as <<NOT LAT P1 30>> the information is hardly worth permanent storage. There are so many other possibilities that information in this form is difficult to utilize. Instead, the positive implications of the statement should be

stored. One of these implications could be to contradict an assertion that was previously believed such as <<LAT P1 30>>. This is a common use of negative statements. Alternatively, a negative statement may be used as a reason for some other belief being contradicted, as for instance in "Did Fred build the Eiffel Tower? No, Fred is not a builder". Thirdly, non-binary negatives often boil down to binary negatives. If I assert that the manager is not in his office, then the meat of the assertion is that the manager is unobtainable which is a binary choice.

## 5.3 Dialogue rules

### 5.3.1 UASSERT and UGOAL

In the last section we said that GOAL, when applied to some goal pattern, would only try to prove the goal pattern and would not try to disprove it. The result would either be "proved" or "unproved". We provide a primitive UGOAL (User Goal) which is used whenever the user wishes to ask the system a question, and which tries both to prove and to disprove the goal statement.

We also provide a primitive UASSERT which is used whenever the user asserts a fact to the program or answers a question put by the program. UASSERT takes care of assertions which it already knows and those which conflict with its knowledge.

We will discuss UGOAL and UASSERT here because they are the basic input functions of the system and must be described before we go on to explain TRY-OR-ASK. Only simple forms of the two primitives will be given although in practice they are more complex. We will explain more of the structure of UGOAL when we consider TRY-OR-ASK.

A flowchart for UASSERT is given in Figure 24 and is more or less self-explanatory. Firstly, the pattern to be asserted is goaled to see if it can already be proved. Secondly, its opposite is goaled to see if there is a contradiction. If neither of these cases hold then the pattern is asserted. So far the approach is the same as that of Coles (1972) in his predicate calculus system. Now we have a difference. Although it might not be possible to disprove the assertion using consequent procedures it may

happen that when we attempt to assert the pattern an antecedent theorem notices an impossibility and fails the assertion. This could happen if we asserted the longitude of P1 and thereby revealed that P1 was in the same position as P2 although they had conflicting properties asserted of them.

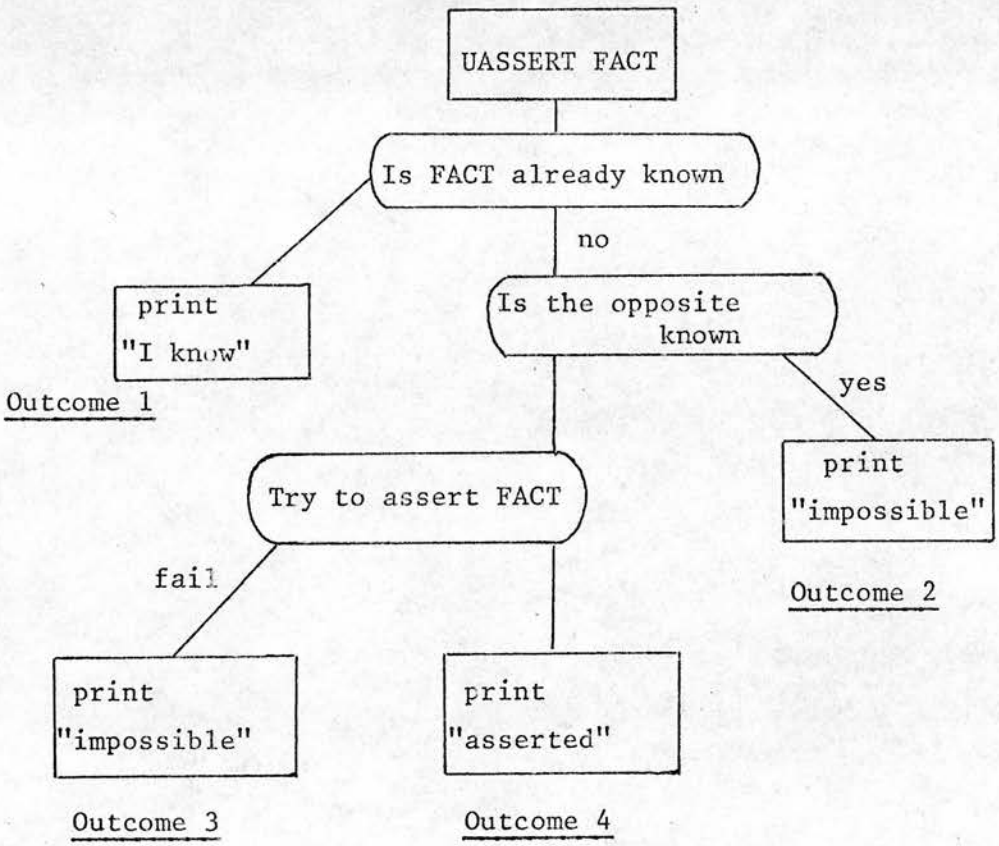



Figure 24

An example dialogue from the program is

```
   USER    : UASSERT <<LAT PLACE1 30>>
PROGRAM      <<LAT PLACE1 30>>ASSERTED    (Outcome 4)
      U    : UASSERT <<LAT PLACE1 30>>
      P      I KNOW                       (Outcome 1)
      U    : UASSERT <<LAT PLACE1 40>>
      P      IMPOSSIBLE                   (Outcome 2)
      U    : UASSERT <<MOUNTAIN PLACE1>>
      P      <<MOUNTAIN PLACE1>> ASSERTED(Outcome 4)
      U    : UASSERT <<LONG PLACE1 30>>
      P      IMPOSSIBLE                   (Outcome 3)
```

The flowchart for a simplified version of UGOAL is given in Figure 25 and is self-explanatory.

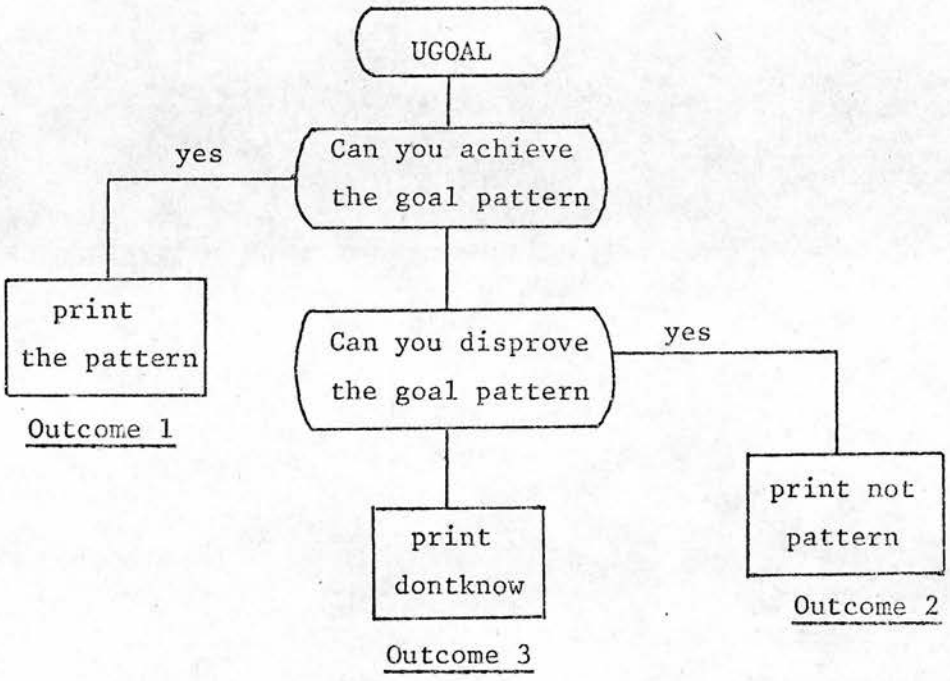



Figure 25

An example dialogue taken from the program is

```
   USER    : UASSERT <<MOUNTAIN PLACE1>>
PROGRAM      <<MOUNTAIN PLACE 1>> ASSERTED
      U    : UGOAL <<LAND PLACE1>>
      P      <<LAND PLACE1>>                    (Outcome 1)
      U    : UGOAL <<SEA PLACE1>>
      P      <<NOT SEA PLACE1>>                 (Outcome 2)
      U    : UGOAL <<LAT PLACE1 £>x>>
      P      DONTKNOW                           (Outcome 3)
      U    : UGOAL <<LAT PLACE1 30>>
      P      <<LAT PLACE1>> ASSERTED
      U    : UGOAL <<LAT PLACE1 £>x>>
      P      <<LAT PLACE1 30 >>                 (Outcome 1)
      U    : UGOAL <<LAT PLACE1 40>>
      P      <<NOT LAT PLACE1 40>>              (Outcome 2)
```

5.3.2 TRY-OR-ASK

Our GEOG-LINE program asked the user questions while it was still in the middle of some computation such as resolving a contradiction or assimilating information. We explained how this gave a goal structure to the dialogue between the user and the program. GEOG-LINE had two reasons for asking questions; either it was curious about some new information or else it needed to resolve a contradiction.

We designed CARTOGRAPHER so that it also asks questions to answer subgoals of questions the user asked it. So, if the user asked if place1 were north of place2 it might ask the user for the latitude of place2 as in the following example.

```
    USER   : UASSERT <<LAT PLACE1 30>>
 PROGRAM     <<LAT PLACE1 30>> ASSERTED
       U   : UGOAL <<NORTH PLACE1 PLACE2>>
       P     TELL ME <<LAT PLACE2 UNDEFINED>>
       U   : <<LAT PLACE2 40>>
       P     <<LAT PLACE2 40>> ASSERTED
             <<NOT NORTH PLACE1 PLACE2>>
```

A question to the user from the program is issued by way of a standard procedure called ASK-FOR which takes care of the reply. The question is always of the form "TELL ME" followed by a pattern. The pattern might have undefined parts which need to be provided by the user, as in the example, or might not, as in the case "TELL ME <<LAND PLACE1>>" which is a simple yes/no question.

The user can reply in any of three ways. He can type in a pattern which matches the request pattern and which fills in any undefined parts. This is an affirmative reply. He can reply in the negative by typing in a pattern which doesn't match the request, as in the reply <<LAND PLACE1>> to the request <<SEA PLACE1>>. Finally, he can reply <<DONTKNOW>>. Any reply the user gives excepting <<DONTKNOW>> is asserted in the data-base using UASSERT, so it can be rejected if it is inconsistent.

An unguided asking of questions as soon as a goal is attempted and failed by the program is too free and we introduce some restrictions. Firstly, we do not make ASK-FOR automatic, but require that it be called explicitly as a particular method of satisfying certain goals. More importantly, if

there are several alternative methods by which the program can achieve a goal we do not want it to ask for the solution to the goal when it has failed on only one method. Rather we wish to try all its methods before it asks at all. We introduce a primitive called TRY-OR-ASK which does this. TRY-OR-ASK evaluates a procedure with a flag called ASK-FLAG set to 0 thus preventing asking and if the procedure fails it tries it again with ASK-FLAG set to 1.

We can illustrate this with the goal pattern <<COAST PLACE1>> . A place is coastal if it is land and if one of its four neighbouring locations is sea. Having shown PLACE1 is on land, we have four ways to succeed on the remainder. One method of determining if any neighbour is on the sea is to ask for its latitude and longitude. This might not be necessary as we may have the assertions

```
<<NORTH PLACE2 PLACE1 10>>
<<EAST PLACE2 PLACE1 0>>
<<SEA PLACE2>>
```

which say that PLACE2 is just north of PLACE1 and is sea. So we want to do

```
TRY-OR-ASK (EITHER GOAL <<JNORTH £>P PLACE1>>; GOAL <<SEA ££P>>;
            ORELSE GOAL <<JSOUTH £>P PLACE1>>; GOAL <<SEA ££P>>;
            ORELSE GOAL <<JEAST  £>P PLACE1>>; GOAL <<SEA ££P>>;
            ORLAST GOAL <<JWEST  £>P PLACE1>>; GOAL <<SEA ££P>>;
            CLOSE)
```

TRY-OR-ASK must also be incorporated into UGOAL. As it was described in the last section, UGOAL PATTERN would try to prove the pattern and then try to disprove it. But this would mean that if the proof of the pattern failed UGOAL would invoke the methods which ask for information from the user. This should not happen until it has been decided that the pattern cannot be disproved without the user's assistance. The order of events we require is

1) Try to prove the pattern without assistance.
2) Try to disprove the pattern without assistance.
3) Try to prove the pattern with assistance.
4) Try to disprove the pattern with assistance.

We arrange this by

TRY-OR-ASK (Try to prove the pattern, otherwise try
to disprove the pattern).

The next example shows the four possibilities

```
    USER   : UASSERT <<SEA PLACE1>>
 PROGRAM     <<SEA PLACE1>> ASSERTED
       U   : UGOAL <<SEA PLACE1>>
       P     <<SEA PLACE1>>                    (proves without asking)
       U   : UGOAL <<LAND PLACE1>>
       P     <<NOT LAND PLACE1>>               (disproves without asking)
       U   : UGOAL <<LAND PLACE2 >>
       P     TELL ME <<LAT PLACE2 UNDEFINED>>
       U   : <<LAT PLACE2 60 >>
       P     <<LAT PLACE2 60 >>ASSERTED
             TELL ME <<LONG PLACE2 UNDEFINED >>
       U   : <<LONG PLACE2 0 >>
       P     <<LONG PLACE2 0 >>ASSERTED
             <<LAND PLACE2 >>                  (proves with asking)
       U   : UGOAL <<SEA PLACE3 >>
       P     TELL ME <<LAT PLACE3 UNDEFINED >>
       U   : <<DONTKNOW >>
       P     TELL ME <<LAND PLACE3 >>
       U   : <<LAND PLACE3>>
       P     <<LAND PLACE3 >>ASSERTED
             <<NOT SEA PLACE3 >>               (disproves with asking)
```

There can be a problem if TRY-OR-ASK is used recursively. In Figure 26 we have a call of TRY-OR-ASK which has four choices of method. The first of these contains a choice of two methods. There is no point here in running the lower call of TRY-OR-ASK in both modes. It may be as well run in the same mode as the higher call.

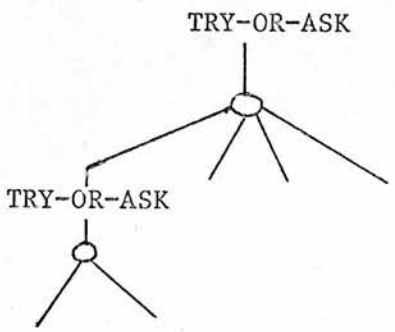



Figure 26

In Figure 27 we have a different situation. Here, the first method is a

conjunction of two goals, as opposed to a disjunction. Now we require the lower TRY-OR-ASK to be run in both modes when the higher one is run in the mode allowing questions.

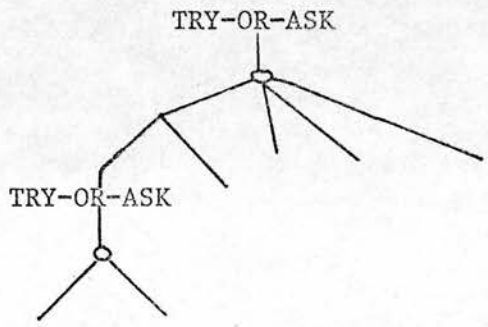



Figure 27

In our system, we have arranged it that if ASK-FLAG is already zero, any lower TRY-OR-ASK will only run its procedure once and will not allow asking, whereas if ASK-FLAG is set to allow for asking, any new TRY-OR-ASK will try both modes. We also have a TRY-DONT-ASK so that the data-base can make adjustments to itself without ever asking the user. We can thus state our dialogue rules so far as

1) Employ the TRY-OR-ASK rules.
2) Otherwise ask as soon as you find something you cannot prove.

If we imagine an AND/OR tree of goals and subgoals which has TRY-OR-ASK attached to various nodes, we see that our two dialogue rules have the undesirable characteristic of asking less important questions first. A more realistic approach would be for the program to ask more important questions first provided it can reasonably assume the user can answer. There are two criteria for choice of question

1) How high up the tree is it, i.e. how significant is it.
2) How much of the tree is below it, i.e. how much work has been saved by asking.

The second criteria is only applicable when the program could have answered the question itself by working it out but suspects that the user

already knows the answer.

It should be becoming clear how intricately connected rules of dialogue are with models of a user's knowledge. In the next section we go into this more deeply.

### 5.3.3 Models of the user

There are certain things we shouldn't ask the user. The following dialogue illustrates an example of this.

```
   USER    :UGOAL <<PLAIN PLACE1>>
PROGRAM    TELL ME <<LAT PLACE1 UNDEFINED>>
      U    :<<DONTKNOW>>
      P    TELL ME <<PLAIN PLACE1>>
```

This is bad form for the program asks the user the question the user asked it. In fact, it is unwise for the program to ask anything it thinks the user does not know. In this case it should realise that the user does not know the answer to his own question since that is why he asked.

The program can get information about the user's knowledge at three times

1) If the user asks a question, it assumes that he doesn't know the answer.
2) If the user makes an assertion, it assumes that he knows that assertion.
3) If it tells the user anything, it assumes that he knows it thereafter.

In our system, we have a very simple method of representing knowledge of the user's knowledge. If we know he knows a pattern, say <<LAND PLACE1>>, then we assert <<KNOW LAND PLACE1>>; and if we know he doesn't know then we assert <<DONTKNOW LAND PLACE1>>. We make these assertions at the times defined above. This method eliminates the most obvious mistakes that the program might otherwise make but has two defects.

First of all, although we can represent statements like "the user knows that place1 is sea" and "the user knows that the latitude of place1 is 30",

we cannot represent "the user knows the latitude of place1". We rectify this by putting "undefined" in the missing part of the pattern. So, if the user asked the program UGOAL<<LAT PLACE1 f>x>>, it would immediately assert <<DONTKNOW LAT PLACE1 UNDEFINED>>. Nor can we represent the desires of the user or his uncertainties. If he asked "is the latitude of place1 30", we should really assert that he thinks it might be 30 and that he would like to know. We have not tackled this problem and so go on to the second one.

The second problem is that mere knowledge of certain facts the user knows and doesn't know is not enough. We need to be able to deduce what he can and cannot deduce from our knowledge of his knowledge and for this we need procedures which we assume he has knowledge of. In the example

```
   USER    : UGOAL <<SEA PLACE1>>
PROGRAM      TELL ME <<LAT PLACE1 UNDEFINED>>
   USER    : <<DONTKNOW>>
PROGRAM      TELL ME <<LAND PLACE1>>
```

the program should not ask its final question. It should assume that if the user knows that place1 was on land, he could work out that it was not on the sea and would not have asked the question in the first place.

Just as consequent and antecedent theorems were necessary to allow the program to keep its data-base consistent, so a model of the user's antecedent and consequent theorems is needed if the program is to keep its model of the user's knowledge consistent. If the user first asserts <<LAND PLACE1>> and then asks <<SEA PLACE1>> the program would have two conflicting pieces of information about the user, namely <<KNOW LAND PLACE1>> and <<DONTKNOW SEA PLACE 1>>. An intelligent system would need procedures to model the user's knowledge so that it could cope with this.

So, two important uses for a procedural model of a user are

1) To keep the model of the user consistent.
2) To know whether it is worth asking the user a question.

The rules we envisage using to coordinate the model of a user are

(a) If we assert that the user knows a fact, then we must erase any facts we had asserted he did not know but which we can now prove he can work out.

(b) If we assert that the user does not know a fact but can prove he does know it, we must resolve this contradiction.

(c) If we wish to find out if a user knows a fact then

i) if we can show he can work it out then we think he knows it.

ii) if we can show he cannot work it out then we think he does not know it.

iii) if, assuming he knew the fact, there is anything we know he doesn't know which we could now show he did know, then he doesn't know the fact.

iv) if, assuming he didn't know the fact, there is anything we know he knows which we could now show he didn't know, then he knows the fact.

Ciii) and Civ) are reductio ad absurdam arguments, based on assuming whether the user knows or does not know the fact and then looking for contradictions between what we think he knows and what we think he doesn't know. Here is an example of their use

```
   USER    : UGOAL <<NORTH PLACE1 PLACE2>>
PROGRAM      TELL ME <<LAT PLACE1 UNDEFINED>>
   USER    : <<LAT PLACE1 30>>
PROGRAM      <<LAT PLACE1 30>> ASSERTED
             TELL ME <<LAT PLACE2 UNDEFINED>>
```

The program should not ask the last question since it knows

```
<<KNOW LAT PLACE1 30>>
<<DONTKNOW NORTH PLACE1 PLACE2>>
```

By using rule ciii) and assuming he knew the latitude of PLACE2, we could show he could work out whether PLACE1 was north of PLACE2 or not. But we know he doesn't know this so he must not know the latitude of PLACE2. So we should not ask him.

It is worth noting that the procedures we need to associate with the user model must be symbolic sometimes. In the example just given we pretend that the user knows that latitude of PLACE2 but we don't pretend the value he knows it to be. We need to know that whatever value he knew it to be he would be able to work out whether PLACE1 was north of PLACE2.

This example also illustrates a failing of our model which is that it lacks foresight. It only knows what it is going to do in the sense that it will do it by following the code. The example shows that the program should also have a model of what it is about to do which it can examine before doing this. Suppose in our example that the program did not know the latitude of either PLACE1 or PLACE2. Then it should realise that it is no use asking the user for the first latitude since it would need to ask him the second also, and in this case the user would have been able to work out the answer to his own question. Even the procedural model we suggest would ask the user the first question and only then realize it should not ask him the second.

If we gave the model of the user the same procedural knowledge as the program itself, then in our arrangement the program would never ask the user any questions. Since there is only one user, the program has received all its factual information from that user and so the user knows at least what the program knows. Moreover, the program knows this. So if the program cannot work out an answer to the user's question, it knows the user cannot help it. Otherwise, the user would have been able to work out the answer itself. So, it seems necessary to introduce some discrepancy between the knowledge of the program and of the user. An interesting project to work on would be one in which a program communicated with several users and attempted to work upon problems with them as a team. This would be an excellent test-bed problem for a dialogue program. We should also realize in such a project that there are many other ways in which a program may decide who and what to ask. If it had a query about pipes then it could deduce that Fred is

worth asking because he is a plumber and in general plumbers know about pipes.

## 5.4 Models of pupils in computer assisted instruction

Now we have considered models of a participant in a dialogue situation, it is a good moment to survey models of pupils in CAI and relate what we have learnt about procedural models to the teaching situation. No CAI program we know of really uses a model of the pupil. Instead, information about the pupil of various kinds is stored in the following ways

1. Statistical information about the student's performance

   Drill and practice programs commonly use this technique to decide when to skip questions or to move on to new concept blocks. The information kept is usually low-level data such as number of correct or incorrect answers given, or the times taken to give these answers. Some programs do also classify types of mistake made. The Edinburgh Arithmetic Project (Howe & Cassels, 1974) uses a knowledge of particular mistakes to generate appropriate new questions.

2. Statistical information about the student's make-up

   This information is usually determined beforehand and compiled into a profile which is a set of scores on various dimensions such as motivation, intelligence, concentration, interest, accuracy and speed. We suspect that a person's psychology cannot be usefully summed up in this dimensional way. Canonical factors cannot be separated and are much too vague to be useful. The missing feature is a mechanism to explain the pupil's behaviour. No profile can provide this as it requires a detailed working model.

3. Data-base models

   Carbonell's program, SCHOLAR, used its data-base of geographical facts to represent the knowledge the user should have at any

time. This is a great improvement on 1 and 2 since it consists of specific information of practical use to the question generator and answer checker in making particular decisions. Unfortunately, no data was kept about the inference rules the pupil knew either correctly or incorrectly. This was probably because Carbonell's data-base made it difficult to deal with these properly.

4. Procedural models

An ideal pupil model would be one such as a human teacher can achieve. A real teacher can know a great variety of facts about a pupil and can make a wide range of inferences about his behaviour, his language use, the misunderstandings he has, how his day-to-day environment affects his work, his interests, and so on. The chief difference between this type of model and those in our classification is that a real teacher knows the kinds of things a pupil does as well as those he knows. The inferences the teacher makes are also concerned with the pupil's acts. If we are to have a model of a pupil which is equally versatile, then the model must be procedural. To use a procedural model to the full a teaching program must be able to discuss, debug and plan corrections for procedures. It must also be able to discover the procedures which make up its student knowledge. Self (1973) gives an excellent discussion of student models, in particular procedural ones, although he does not deal with these aspects.

We have seen uses of a model of a user in a dialogue situation with CARTOGRAPHER. We will conclude with an example taken from a teaching situation. An eager pupil will try out hypotheses for himself. Knowing the possible hypotheses and using evidence from the dialogue to decide which of these the pupil holds will enable the teacher to react more appropriately. Suppose, in Figure 28, the pupil asked the temperature of places 1 and 2, and found they were the same. He might decide to test the hypothesis that temperatures at the same latitude are the same.

If he tested this on

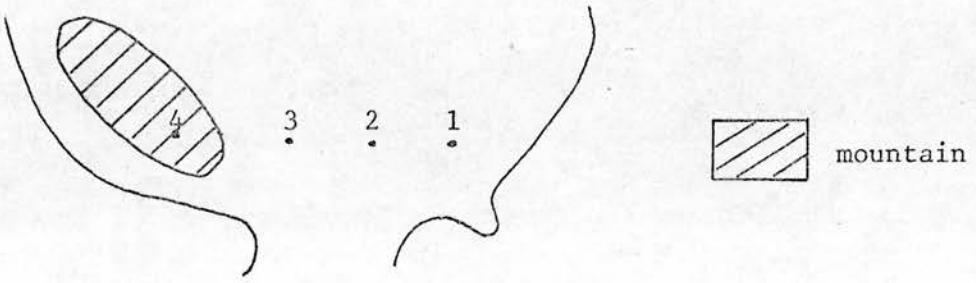



Figure 28

place3, the computer should realise that this is what the pupil is doing. Then when the pupil tried place4 the program could state the temperature of 4 and add that it is on a mountain so as not to spoil the pupil's hypothesis.

Procedural models of the pupil's expertise and his learning behaviour raise a very interesting problem of inference. This problem is about how a program, in this case the teaching program, can discover what procedures make up the pupil's behaviour and explain it. In the case of simple factual models of a pupil, the program could both remember things it has told the pupil and the pupil's questions and answers. This situation only becomes complex when the pupil's inference rules are taken into account. If the program assumes the pupil either knows a rule or doesn't know it, with no possibility of his having an incorrect version, then it can infer that the pupil does not know any combination of facts and rules which would allow him to deduce a fact which the program knows the pupil doesn't know.

Since the pupil can have incorrect versions of rules and procedures the problem is even more difficult. Somehow, the program must know the possible variations of each rule. These could be in the form of sets of complete incorrect versions or of a correct version coupled with a knowledge of possible bugs. There are two ways in which the program can now make its inferences. The forward way is to take the pupil's behaviour and notice properties of it which, from past experience, directly imply possible bugs.

The backward way is to hypothesize possible procedures or bugs for the model pupil and then run the model to see if its predicted behaviour matches the pupil's behaviour at all. This is a good example of the use of procedural models as simulations for deducing the state of affairs in some system.

Whatever the method employed, the problem seems to be central to AI and not merely CAI. One of the impressive abilities of people is that of postulating possible mechanisms for systems which have certain behaviours. It is almost certain that we have many special-purpose procedures to help us in this task. This is an area which will need much more research before we can build effective teaching programs.

## Chapter 6 A program which answers "why" questions

### 6.1 Introduction

Our third illustration of the principles put forward in Chapters 1 to 3, is a program to answer "why" questions. This introduction classifies "why" questions to set up a context in which we can see more exactly what our program deals with. The next section describes a programming method which allows us to build expressions which can be both evaluated and easily manipulated. The method allows us to express "why" questions so that our program has an elegant means of answering them. We use it to implement a LISP-like language as a basis for our main program. Then we give a description of the program with examples of output, and in the final section we point out the very strong relationship between this program and two themes which occurred earlier, namely the distribution principle and the idea of clusters of functions. The program is a strong, concrete example of these two ideas.

We can classify "why" questions into four types

1. Definitional why questions.

   Here the answer follows immediately by expanding the definition. In the example "Why is Malta an island?", we could reply "Because it is surrounded by water". There is no complicated deduction, so the only rule applied is explained.

2. Purpose why questions.

   These are the kinds of questions answered by Winograd's program about its own actions. For example, the question "Why did you pick up the red pyramid?" is answered by "To put it in the box".

3. Causal why questions.

   These ask for explanations in terms of a working model or a mechanism. For example "Why did the green ball go in the pocket?" could invoke the explanation "Because the white ball

hit it to the blue one which deflected it into the pocket".

4. Deductive why questions.

These are similar to type 3 questions in the kind of explanation given, but in place of a mechanism we have to consider the working of a deduction procedure. For example, questions like "Why are opposite angles of an isosceles triangle equal?" are roughly equivalent to "How do you know?" questions.

If we consider deduction as a particular type of process and consider the execution of a program as a generalisation of deduction, then these four classes can be related. In type 1, the process is trivial, in 2, it is the problem solving process of a goal directed robot, in 3, it is a physical mechanism and in 4, it is a process of logical deduction. Our classification is semantic, since many question forms can be answered in ways appropriate to several of our classes. For example, the definitional question "Why is Malta an island?" can be treated causally and given a geophysical explanation; and "Why do porcupines have quills?" can be treated biologically, as a causal question, or evolutionarily as a purpose question. Deciding which way to treat a why question is a possible source of ambiguity.

When answering "why" questions, we often use a tree of explanations. If the temperature of a place depends on two factors, the heat arriving and the heat being lost, then a statement of these factors at a particular place, together with a statement of the relationship between them, would constitute a first order explanation for the value of the temperature at that place. However, we could go deeper into the explanation by asking about either of the two factors. We would produce a tree. In Winograd's program, the robot had a goal tree. Each goal action had subgoal actions. The reason for executing a subgoal is to achieve its main goal. Winograd's program answered type 2 questions.

Notice that type 3 and 4 questions are different from type 2 questions. Given a node, A, below a node, B, in a goal tree, the answer to the type 2 question "why did you do A?" would be "in order to do B" whereas the type 3

question "Why did B happen?" would be answered "Because A did". One works up the tree, and the other works downwards. The first explanation is in terms of the function of the node in the tree and the second is in terms of the structure of the node.

In our program, we deal with simple type 4 questions such as "Why is factorial 4 positive?" and "Why is the temperature of "A" greater than the temperature of "B"?". The program gives explanations which go progressively deeper into the explanation tree for the original question. This is as though we asked the program "Why? but why? but why?", and so on.

How relevant replies might be given to a "why" question is an extremely interesting problem although our own program treats every part of an explanation on an equal status. There are two possible lines of attack for this problem.

Firstly, the user's knowledge must be taken into account. It is unnecessary to tell him parts of an explanation which he already knows. This involves more than a comparison of facts, for consider the question "Why is the temperature at A, 50?". The program can evaluate the temperature and use its evaluation to produce an explanation. However, the parts of this explanation which the user already knows are those which he can work out himself. A good program needs a procedural model of the user's knowledge, as argued in Chapter 5. It might even turn out that the user can do the evaluation completely but incorrectly. He might know that the temperature is 50 but evaluate it himself to 60. If the program could model the user's evaluation, it could pin-point the error and would have found a relevant explanation.

There are also parts of an explanation which are naturally more relevant than others, regardless of the user's knowledge. The mechanism in Figure 29 allows a ball to roll from the top, via various tubes and ramps, into either pan A or pan B. A full explanation for the question "Why did the ball land in A" would contain much irrelevant detail. If the hidden comparison were noted (why pan A rather than pan B) then the relevant feature, namely the switch setting, would be highlighted.

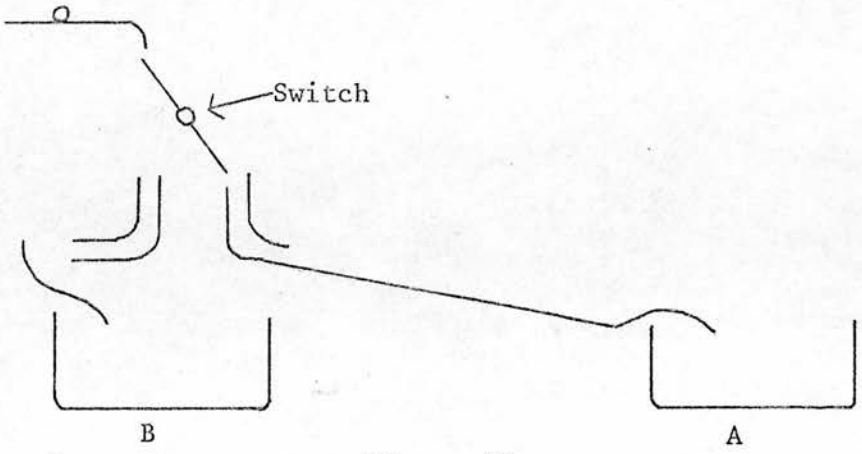



Figure 29

This discussion of "why" questions only touches on some of the problems. It is intended as a setting for our program whose purpose is to illustrate a programming method for our theory of the distribution of knowledge. Brown (1973) has investigated "why" questions in a tutorial setting. He wrote a program to teach electronic troubleshooting. As our aims are different, we do not describe his program but simply refer the reader to it.

## 6.2 Using closures to build structured functions

In this section, we describe the programming method by which we write our program to answer "why" questions. Firstly, let us see why we need the method.

Although it is easy to see analogies between computer languages and natural languages, there are important differences which limit the analogies' application. Computer languages use uniformly structured, recursive, and generally context-free grammars. The signalling system, for example, brackets, is extremely nested. Natural languages are more complex than this and are very knowledge and context dependent.

For example, relations, predicates and functions in natural language either take an indefinite number of arguments, the number depending on many factors best known to the relations themselves, or else they take the context itself as one argument. The methods are more or less equivalent. Consider the predicate "big". If we consider it to take one argument which is a noun, it

must take different meanings according to the noun. A big flea is different from a big dog. However, "big spanner" needs to know about the use of the spanner and the nut it is to be applied to. In general, "big" may need arbitrary information from the context.

There is also a complicated system for referencing in natural language, as we showed in Chapter 4, and many semantic structures are governed by the process of metaphor. Natural language is far more subtle than computer languages.

One particular difference concerns us here and we will describe it by means of an example. Note that an expression in POP-2 or in LISP given to the evaluator is an English-like description of a task to be done. For example, "is the temperature of PLACE1 greater than the temperature of PLACE2?" can be rendered in POP-2 as

GREATER (TEMPERATURE(PLACE1), TEMPERATURE (PLACE2)).

To answer the question in this example, we might ask ourselves "how do I find out if the temperature of a place is greater than that of another?", perhaps deciding to compare the latitudes of the two places. The way we use natural language allows us to do this kind of reasoning, because it allows us to symbolically manipulate natural language expressions and consider such sub-expressions as "the temperature of PLACE1". We can now state the problem as follows

> In most programming languages the structure implicit in an expression is only accessible to the compiler which parses the expression.

The problem lies in the execution time of the sub-expressions of a structure. When we execute the statement

GREATER (TEMPERATURE (PLACE1), TEMPERATURE (PLACE2))

the temperature of PLACE1 is evaluated first of all, secondly the temperature of PLACE2 is evaluated, and finally GREATER is evaluated with the values of the temperatures as its arguments. GREATER has no chance to examine the structures which produced these values. It can never know what its arguments

represent and can, therefore, never employ any special knowledge about how to find out if one place has a greater temperature than another. It is restricted to finding a relationship only between numbers.

Since LISP procedures are manipulable list structures which map directly onto the corresponding text, it is possible to arrange for a procedure to be given its arguments unevaluated. Certain procedures called special functions behave like this (McCarthy et al, 1962), and in standard LISP systems there are ways to produce new special functions although it is more usual to use functions which receive evaluated arguments.

We have used the closure function facility which is present in POP-2 and certain other languages to write a version of LISP where the natural mechanism of passing arguments to a function leaves the arguments unevaluated. We will describe this version of LISP fully in the next section and here we introduce the implementation method. We refer to this method as the method of using closures to produce structured functions.

In POP-2, a closure function consists of a function combined with several items, and it can be applied to arguments in exactly the same way as can a normal POP-2 function. When this happens, the items associated with the function are added to the stack of arguments already given. The process of attaching items to a function to produce a closure function is called freezing in the items. If we freeze in n items to a function f to produce a closure F, and then apply F to m arguments, f gets automatically applied with m + n arguments. (For further information about closures see Appendix 1.)

To build up structured functions using this facility we let the frozen values of a function F be functions themselves. F is the cement which binds together these values. At certain points in its execution, F evaluates its arguments and uses them as subroutines. The function CONDITION can be used to make structured functions.

```
FUNCTION CONDITION FTEST FTRUE FFALSE;
IF .FTEST THEN .FTRUE ELSE .FFALSE CLOSE
END;
```

CONDITION is a function taking three arguments. It applies its first and and if this is true, it applies the second, otherwise it applies the third.

We can use CONDITION to make a structured function by freezing into it the three functions DECIDE, PR(%2%), and PR(%3%); DECIDE is a function which evaluates to TRUE or FALSE. Decorated brackets freeze in parameters so PR(%2%) is the printing function combined with the item 2. It is a function to print 2. The structured function we produce is

```
CONDITION(%DECIDE,PR(%2%),PR(%3%)%)
```

When this is executed, CONDITION will bind FTEST, FTRUE, and FFALSE to its three arguments which are unevaluated functions. It will then evaluate DECIDE and depending on the result will evaluate either PR(%2%) or PR(%3%).

CONDITION, used in this way, is an example of a concept dual to subroutines. In the case of subroutines, many routines may reference any one subroutine so the code for that subroutine may be used in many places. If we consider the routines which use subroutines to be templates then we see that no template may be used twice with different sets of subroutines. We invert this in our use of frozen values. Each basic function like CONDITION is a template with vacant slots which can be filled in many ways by producing closure functions of it.

It is important to notice that this technique allows us to build hierarchical structures which can be both evaluated and examined, just as LISP list-structured functions may be both interpreted and manipulated.

Using closures to produce structured functions allows us to produce a manipulable and executable structure corresponding to

```
GREATER(TEMPERATURE(PLACE1), TEMPERATURE(PLACE2))
```

and whose substructures, rather than their values, are passed as arguments to their combining templates. When

```
GREATER(%TEMPERATURE(%PLACE1%), TEMPERATURE(%PLACE2%)%)
```

is evaluated, GREATER is given the two arguments

```
TEMPERATURE(%PLACE1%) and TEMPERATURE(%PLACE2%)
```

It can evaluate these, but it can also examine them first. We can thus attach extra knowledge to GREATER by allowing it to examine its arguments. An example of an enhanced version of GREATER is shown below.

```
FUNCTION GREATER FN1 FN2;
The arguments of GREATER are called FN1 and FN2
If FN1 is of the form
     TEMPERATURE(%X%)
and FN2 is of the form
     TEMPERATURE(%Y%)
then evaluate
     LATITUDE(X) <LATITUDE(Y)
otherwise evaluate FN1 and FN2 and see
     if the first result is greater than the second
END;
```

GREATER checks if its arguments are temperatures and, if they are, compares the latitudes of the places the temperatures are of. Otherwise, it evaluates its arguments and compares the results.

Now that we have an accessible structure corresponding to a command, we find we also have structures corresponding to facts. Any program which evaluates to true is such a structure. As an example, consider

```
EQUAL(%TEMPERATURE(%PLACE1%),70%)
```

which would be true if the temperature of place1 is 70. An advantage of having this structure is that we can ask why a fact is true.

```
WHY(EQUAL(TEMPERATURE(PLACE1),70))
```

would have evaluated to WHY(TRUE). This does not help WHY to answer the question, as opposed to the following example.

```
WHY(EQUAL(%TEMPERATURE(%PLACE1%),70%))
```

which does. Note that we have not adhered to the method of closures in the last example. Since all functions should evaluate their arguments, the argument 70 should be a function which evaluates to 70, i.e. IDENTITY(%70%).

This presents the same problem for we then wish IDENTITY to evaluate its argument. There seems to be no bottom level but we resolve the problem in the LISP-like language we describe in the next section.

## 6.3 A LISP-like system using closures

In the following section we assume that the reader is familiar with the LISP 1.5 language (McCarthy et al, 1962).

### 6.3.1 Structured expressions

In LISP 1.5, expressions can be evaluated in an environment and are list structures composed of list cells. The structures can be accessed as data-structures by use of CAR and CDR. In our system, expressions are structured functions produced by freezing sub-expressions into certain standard functions. These functions are our LISP combinators and are as follows

| | | |
|---|---|---|
| LAMBDAFN | QLAMBDAFN | APPFN |
| CONDFN | WORDFN | VALUE |
| NUMBERFN | VALFN | BINDFN |

Thus the POP-2 structured expression corresponding to

```
((LAMBDA (X) (PLUS X X))2)
```

is

```
APPFN(% LAMBDAFN(%[X],
                 APPFN (%VALUE(%"PLUS"%),
                        [%VALUE(%"X"%),
                          VALUE(%"X"%)%]%)%),
        NUMBERFN(%2%)%)
```

(N.B. Square brackets and decorated square brackets denote lists. See Appendix 1)

Since structured expressions written in POP-2 are cumbersome, we need simple syntactic ways of manipulating them. In particular, we provide ways of writing them as though they were LISP expressions; ways of testing and accessing structured expressions built from standard combinators; ways of matching structured expressions so that variables can be bound to

required subexpressions; and ways of building expressions from subexpressions produced by matching. These facilities provide powerful ways of handling structured expressions and are used in the examples given later.

There is a big difference between the way in which a LISP 1.5 expression is evaluated in an environment and the way in which a structured expression is evaluated in an environment. In LISP 1.5 the evaluation is done by an evaluator which knows all the types of structure it is likely to come across and knows how to evaluate each of them. In the sense of Chapter 3, the evaluation is extensional in that expressions are data which have evaluation done to them. On the other hand, a structured expression knows how to evaluate itself and evaluation is intensional. If we ask a structured expression to evaluate itself, it will itself determine when to ask its subexpressions to evaluate themselves. It might never evaluate them but might simply examine their structure. For example, the result of

```
(GREATER(PLUS X 2) (PLUS X 1))
```

is true whatever X, so X need not be evaluated.

Since each function in a structured expression is evaluated by the POP-2 evaluator, it is easy to confuse the LISP-evaluation and the POP-2 evaluation and feel that structured expressions are evaluated in no way differently to LISP expressions. However, in a standard LISP interpreter, the word LAMBDA is simply a constant dealt with by the interpreter, whereas in our system LAMBDAFN is a function in which we locate particular expertise for evaluating LAMBDA expressions. In our system it would be possible to introduce any object as part of a structured expression provided it behaved correctly when asked to evaluate itself. We have distributed knowledge about interpretation throughout the system by making all structures into procedures. This is an instance of the distribution principle. Later we will come across other types of knowledge which we distribute through the system.

Since a structured expression is a POP-2 function as it is produced as a closure of ther POP-2 functions, it can be applied like any other POP-2 function. POP-2 application of a structured expression corresponds to LISP evaluation in an environment. If

```
((LAMBDA (X) (PLUS X X)) 2)
```

were evaluated in an environment where PLUS was an addition operator, by applying the corresponding POP-2 structured expression given earlier, the result would be the structured expression corresponding to 4 which is NUMBERFN(%4%).

All structured expressions can be evaluated in this way. Some, such as NUMBERFN(%4%) just return themselves while others do more complex evaluation. Some structured expressions can also be applied. This is done by APPFN which corresponds to LISP function application. So (MINUS 2 1) corresponds to

```
APPFN(%VALUE(%"MINUS"%),[%NUMBERFN(%2%),NUMBERFN(%1%)%]%)
```

When this POP-2 expression is evaluated, the function APPFN is executed with its two arguments being the items frozen into it. It evaluates the first argument, VALUE(%"MINUS"%), which results in the primitive function MINUS and then it applies this function, MINUS, to the arguments NUMBERFN(%2%) and NUMBERFN(%1%).

Since the structured expression (LAMBDA (X) X) represents a function, it can be applied to arguments by APPFN. APPFN will first evaluate it so a LAMBDA expression when evaluated must return itself.

### 6.3.2 The Macro, LISP

We now examine the behaviour of the primitive LISP combinators that we provide, and give the syntax rules which allow us to easily describe structured expressions built from the combinator. A description of an expression in LISP-like syntax is preceded by the word LISP. LISP is a macro which reads the description and builds an expression corresponding to it.

The format of our description of each primitive combinator is as follows. We first describe the behaviour of that combinator and then we give its syntax rules. The syntax rules are explained by showing the transformation from the LISP expression to the resulting POP-2 expression and then by

giving an example fo this transformation. We use < > to denote metasyntactic classes, and * to denote any number of consecutive elements of the class preceding *, <<class>> denotes the class of POP-2 structured expressions corresponding to the syntactic class <class>. For example, if "A" is a LISP expression from the class <expression-1>, then "value (%"A"%) is the corresponding structured expression from the class <<expression-1>>.

NUMBERFN takes one argument which is a number. A structured expression made from NUMBERFN has one frozen value and this can be accessed by NUMOF. We test if an object is a structured expression formed from NUMBERFN with the primitive ISNUM. NUMBERFN(%n%) evaluates to another instance of NUMBERFN partially applied to n i.e.

APPLY(NUMBERFN(%N%)) = NUMBERFN(%N%)

The syntax for NUMBERFN is as follows

LISP <number> is rewritten as
NUMBERFN(%<number>%), for example
LISP 2 becomes
NUMBERFN(%2%)

WORDFN takes one argument which is a word. The accessing function is WORDOF and the testing predicate is ISWD. WORDFN(%<word>%) evaluates to WORDFN(%<word>%). The syntax for WORDFN is as follows

LISP "<word>" is rewritten as
WORDFN(%"<word>"%), for example
LISP "A" becomes
WORDFN(%"A"%)

VALUE takes one argument which is a word. The accessing function is IDENTOF and the testing predicate is ISIDENT. VALUE(%"A"%) evaluates to the current value of "A".

The syntax for VALUEFN is as follows

LISP <identifier> is rewritten as
VALUE(%"<identifier>"%), for example
LISP A becomes
VALUE(%"A"%)

CONDFN takes three arguments called the condition, the first branch, and the second branch. When evaluated it evaluates its condition and depending on whether the result is true or false, it then evaluates its first or its second branch. The accessing functions are CONDOF, FSTBROF, and SNDBROF. The testing function is ISCOND. The syntax for CONDFN is as follows

```
LISP(COND<expression-1><expression-2><expression-3>) is rewritten as
CONDFN(%<<expression-1>>,<<expression-2>>,<<expression-3>>) so
LISP COND(TEST 0 1) becomes
CONDFN(%VALUE(%"TEST"%),NUMBERFN(%0%),NUMBERFN(%1%)%)
```

APPFN takes two arguments called the function and the arguments. The function is a structured expression and the arguments are a list of structured expressions. The accessing functions are FUNOF and ARGSOF and the testing function is ISAPP. When evaluated it evaluates its first argument and asks it to apply itself to the expressions in the list.

The syntax for APPFN is as follows

```
LISP(<expression-1><expression-2> ... <expression-n>) is rewritten as
APPFN(%<<expression-1>>,[%<<expression-2>> ... << expression-n>>%]%)
LISP(MINUS 2 1) becomes
APPFN(%VALUE(%"MINUS"%),[%NUMBERFN(%2%),NUMBERFN(%1%)%]%)
```

LAMBDAFN takes two arguments called the parameters and the body. The parameters are a list of words and the body is a structured expression. The accessing functions are PAROF and BODYOF, and the testing predicate is ISLAMBDA. When evaluated it returns itself frozen to its two arguments.

When APPFN asks a LAMBDA structured expression to evaluate itself in some environment and with some arguments, the LAMBDAFN of the expression first evaluates its arguments in the environment and binds the results to the parameters which are local to the LAMBDA. It then evaluates the body of the expression within the new environment created by the bindings. This is an example of the call-by-value method of argument passing.

The syntax for LAMBDAFN is as follows

```
LISP(LAMBDA(<word>*)<expression>) is rewritten as
LAMBDAFN(%[<word>*],<<expression>>%), for example
LISP(LAMBDA (X Y) X) becomes
LAMBDAFN(%[X Y], VALUE(%"X"%)%)
```

QLAMBDAFN takes two arguments called the parameters and the body. The parameters are a list of words and the body is a structured expression. The accessing functions are PAROF and BODYOF, and the testing predicate is ISQLAMBDA. When evaluated, it returns itself frozen to its two arguments. A LAMBDAQ expression behaves in the same way as a LAMBDA expression except that QLAMBDAFN binds its arguments without evaluating them. This is an instance of the call-by-name method of argument passing. In our examples the names will often be complex structured expressions. The syntax for XLAMBDAFN is similar to that for LAMBDAFN.

```
LISP(LAMBDAQ(<word>*)<expression>) is rewritten as
QLAMBDAFN(%[<word>*],<<expression>>%), for example
LISP(LAMBDAQ (X Y) X) becomes
QLAMBDAFN(%[X Y],VALUE(%"X"%)%)
```

It is important to notice that usually a function will receive its arguments unevaluated because APPFN does not evaluate them. This means that the function is free to examine the arguments and may bring to bear any special knowledge it has to help it in its evaluation. This property allows us to deal with the problems discussed in the last section. Of course, the function might evaluate its arguments immediately but the onus is on the function and not on APPFN to decide.

We have two primitives for producing new functions; LAMBDAQ which takes quoted arguments, and LAMBDA which takes its arguments unquoted. APPFN works in the same way for each of them.

For example

```
((LAMBDA (X) X)(PLUS 2 2))
```

will evaluate to 4 (that is to the expression NUMBERFN(%4%)).

and

```
((LAMBDAQ (X) (X))(PLUS 2 2))
```

will evaluate to 4.

The QUOTE function of LISP 1.5 is not necessary in our system since arguments are naturally quoted. The onus is on the functions to decide what they want to do with their arguments. There is a case occasionally for the opposite of QUOTE called EVALUATE. Suppose we have a function FQ which normally takes its argument unevaluated and after examining it returns as result as structured expression. FQ may be a function to delete double negatives for example so

(FQ(NOT (NOT X))) becomes X

How can we write a function F which takes its argument, evaluates it and applies FQ to the result? Suppose we wished to remove four negatives by saying

(F(FQ (NOT(NOT(NOT(NOT X))))))

We cannot write F as

(LAMBDA (X) FQ (X))

because since FQ takes its argument unevaluated it will try to remove a double negative from X and not from its value. If we had an override called EVALUATE, as suggested above, we could say

(LAMBDA (X) FQ (EVALUATE(X)))

so that, in this example, the inner FQ would be evaluated and this FQ would receive the result as its argument.

### 6.3.3 Other primitives

We have only a few primitives in our LISP-like system, such as MINUS, PLUS, TIMES, DIVIDE, GREATER, LESSER, ZERO, AND, OR, NOT, EQUAL, NEGATIVE, POSITIVE, SETQ, and EVAL. Most of these primitives are self-explanatory and we only need to explain SETQ and EVAL.

SETQ takes two arguments. It expects the first to be an expression of the form VALUE(%"<identifier>"%) and will evaluate its second argument and assign the result to the identifier. This is useful for defining functions as in

```
(SETQ DOUBLE (LAMBDA (X) (PLUS X X)));
```

EVAL is similar to LISP and is followed by a LISP expression. Just as LISP results in a structured expression so does EVAL. EVAL first applies LISP to produce an expression and then evaluates that expression. The result of the evaluation is the structured expression given as the result of EVAL. For example

```
EVAL (PLUS 2 2);
```

results in the structured expression 4.

### 6.3.4 Matching

One of the main uses of the system is for writing functions which make use of the symbolic structure of their arguments to evaluate themselves or to do other things such as paraphrase themselves in English or give reasons for themselves. So we can say

```
EVAL (PRINTOUT(WHYQ (POSITIVE 3)));
```

which would result in the printout

```
"3 is greater than 0"
```

WHYQ has used the structure of its argument to produce a structured expression standing for the reason, and PRINTOUT has used the structure of this to print out the English. This is a very simple case but in general we need to do complex manipulations on structured expressions.

Pattern matching techniques as used in SNOBOL (Farber et al, 1964) or PLANNER are ideal for manipulations of structured expressions. The primitive MATCH will take two structured expressions and match them in a top-down left to

right manner. It will return the result true or false. So

```
MATCH (MINUS 2 2) (MINUS 2 2);
```

will return true.

Since one of the purposes of matching is to extract subexpressions of an expression, we introduce a new type of combinator called BINDFN. LISP :A; will result in BINDFN(%"A"%). If an instance of BINDFN is reached on the right hand side of a match, the expression being matched with it will be assigned to the variable whose identifier occurs in the BINDFN expression. So

```
MATCH (MINUS 2 1) (:F :A :B);
```

will return true and bind F to MINUS, A to 2 and B to 1. If a match returns false all its bindings are undone. The prefix ":" may only be used in the right hand expression of a match statement as our matching is one-way.

It is also possible to use values of variables in matching by using the prefix "/" instead of ":". So after the above match

```
MATCH (MINUS 3 4) (/F 3 4);
```

will return true.

Two other prefixes "::" and "//" are provided. These match the entire list of arguments of an application. The difference between them is the same as between ":" and "/".

```
MATCH (MINUS 2 1) (MINUS ::ARGS);
```

followed by

```
MATCH (MINUS 2 1) (MINUS //ARGS);
```

will both return true.

Having accessed the required subparts of an expression we need to use these in building up new expressions. Suppose we had two expressions in A and B. The expression

```
LISP (MINUS /A /B);
```

describes a structured expression built up using the current values of A and B. If A were the expression (PLUS 1 1) and B were 1, then the new expression would be the same as

```
LISP (MINUS (PLUS 1 1) 1);
```

The matching primitives described here greatly simplified our program for answering "why" questions.

## 6.4 Answering "why" questions

In the last section, we gave an implementation of a LISP-like system which has the property that arguments of a function are given to the function without being evaluated. Using this property, we can represent "why" questions as unevaluated arguments given to a function WHYQ. For example, "why is the temperature of A greater than the temperature of B?" could be represented as

```
LISP(WHYQ(GREATER(TEMPERATURE "A")(TEMPERATURE "B")));
```

Since, as its argument WHYQ receives a structure representing a fact, it has the information it needs to give an explanation of that fact.

Our program is implemented in the LISP-like language we have described. Just as in this language, knowledge about evaluation is distributed through the system as procedures attached to the various components of structured expressions, so we distribute knowledge about answering why questions. This knowledge is in the form of procedures attached to functions and to primitive combinators and the procedures are called "why-functions", abbreviated as WHYFNS. So, knowledge used to explain why an expression evaluates to be positive is associated with the function POSITIVE and is the procedure called the why-function of POSITIVE or (WHYFN POSITIVE). When we wish to

show that a particular expression is positive we give that expression as argument to the why-function of POSITIVE. We give examples later.

The WHYFN of F will very often use the same sub-routines as F. Suppose for example that the temperature of a place was the sum of a basic temperature depending on latitude, and of two other factors due to presence of coasts or of mountains. Then the WHYFN of TEMPERATURE could find the answer to "Why is the temperature of PLACE1 70?" by evaluating the three factors and giving an explanation in terms of them. Such an explanation could be "Because there is no coastal or mountain effect and the basic temperature is 70".

By calling the whyfns of the sub-routines we can get a deeper explanation of the fact, including an explanation of the zero coastal and mountain effects. This produces the kind of explanation tree referred to in Section 6.1. It also introduces problems of organisation for a complete step by step explanation of even a simple fact like this one is often extremely long and filled with trivial steps. The choice of a relevant reply depends on other knowledge than simply this method of extracting a reason. It depends on context in a discourse, the intentions of the asker and knowledge of the asker's knowledge. This matter was raised in Section 6.1 and here we use a simple organization without these extra sources of guidance.

Because WHYFN(F) utilizes the sub-routines of F, and also calls the whyfns of those sub-routines, there is a clear relationship between the evaluation of a program representing a fact and the production of a reason for it. The structure of the reason follows the structure of sub-routine calls of the program. There is a similar relationship between a reason for a fact and deductions which prove the fact true. If we prove the classic statement "Socrates is human" by proving first "Socrates is a Greek", and second, "All Greeks are human" then the two subgoals in our proof provide, by themselves, a reason for Socrates' humanity.

Consider now the relationship between WHYFN(F) and F. Have we not cheated a little by providing the whyfn of a function by ourselves? It would certainly be more general if we had a mechanical way to generate whyfns. In general, this problem is very deep and involves proving theorems about programs. However, in certain simple cases we can exploit the relationship

between execution and deduction to avoid the need to produce an explicit whyfn. Consider a function F. We only needed to provide a whyfn for it because we treated it as an indivisible entity. In fact, the function is structured. If it were a LISP function it would be represented by a list structure definition. The function has the effect of the list structure substituted in its place. For instance

```
(DOUBLE 2)
```

is equivalent to the expression

```
((LAMBDA(X) (PLUS X X))2)
```

produced by replacing the structured definition of DOUBLE for DOUBLE. If we write function definitions using our method of closures, e.g.

```
LISP (LAMBDA(X) (PLUS X X));
```

then we have precisely the form needed for answering why questions. We can answer

```
(WHY (EQUAL (DOUBLE 2) 4))
```

without needing any why-functions attached to DOUBLE but by expanding it into its definition.

This sketches out our implementation of why-functions. We must now go over the same ground in far more detail giving examples of output from our program as we go. Let's begin with some simple examples. Suppose we type into the system

```
EVAL (WHYQ (IS-SEA "B"));
```

then the result is the reason "TRIVIAL". WHYQ saw that its argument was the expression (IS-SEA "B") so it looked for (WHYFN IS-SEA) and applied this to the expression "B". (WHYFN IS-SEA) is a particularly simple procedure which always returns the reason "TRIVIAL".

In this case, the why function of IS-SEA did not examine its argument. In general, it is important that arguments are unevaluated since most why-functions will examine them. We can see this with a slightly more complex case. Suppose we had typed

```
EVAL (WHYQ (NOT(IS-SEA "A") )).
```

In this case, the WHYFN of NOT would be used and it would be given the argument (IS-SEA "A"). This time there is usable structure. (WHYFN NOT) will examine this structure by saying

```
MATCH /S (:F ::ARGS)
```

where S is the structure. The match will succeed and F will be bound to IS-SEA. All that (WHYFN NOT) has to do now is to find the opposite of F by saying

```
(OPPOSITE F)   G;
```

and then to return with the reason

```
LISP (/G //ARGS);
```

which is of course the expression

```
(IS-LAND "A").
```

So, "A" is not on the sea because it is on the land.

Now, we can associate why-functions with any predicate since an expression whose top level function is a predicate can be the subject of a why question. Other functions such as PLUS cannot have why-functions since it is meaningless to say "why is 2 plus 2?". Instead, PLUS can have other functions attached to help answer why questions. For example, the expression (PLUS 2 2) may know how to show why it is positive rather than why it is true. We go into this later as it occurs in many similar examples and is an illustration of the distribution principle.

Adding why-functions to predicates is one way to increase the system's ability to answer why questions. Some simple why-functions we have in our system are for the predicates NOT, OR and AND. We also have why-functions for LAMBDA and COND which we discuss later.

(WHYFN NOT) examines its argument to see if it is the expression "FALSE". If so, it returns "TRIVIAL". Otherwise, it matches the argument with (:F ::ARGS) and finds the opposite, G, of F. It then returns LISP (/G //ARGS). If the match gives false then (WHYFN NOT) will return "DONTKNOW".

(WHYFN AND) is given two arguments and needs to return a reason for the conjunction of the two. It therefore returns the conjunction of the reasons of the two. Thus

```
WHYQ (AND FACT1 FACT2) = (AND REASON1 REASON2)
where      REASON 1 = (WHYQ FACT1)
           REASON 2 = (WHYQ FACT2)
```

WHYQ is propogated through the AND, and the result is another structured expression. We do some simplification on the result. If either REASON1 or REASON2 is "DONTKNOW", the total result is "DONTKNOW"; and if either is "TRIVIAL" it is removed from the result leaving only non-trivial reasons.

(WHYFN OR) is given two arguments and first finds out which of them are true. It returns the reason for a true one.

To give an example of these why-function in action consider if we typed

```
EVAL (WHYQ (AND(IS-LAND "A") (NOT(IS-SEA "D"))))
```

The result would be

```
(AND (NOT(IS-SEA "A")) (IS-LAND "D"))
```

There are two ways we can improve upon these simple why-functions. First of all, there are many special symbolic rules which would enhance them. For example, the answer to the question "Why is either A on land or B on the sea?" typed in as

```
EVAL (WHYQ (OR(IS-LAND "A") (IS-SEA "A")))
```

should be "TRIVIAL" since IS-LAND is the opposite of IS-SEA. This is easy to introduce. Suppose the arguments of (WHYFN OR) are ARG1 and ARG2. All it needs to do is the check

```
 If MATCH /ARG1  (:F ::ARGS1)
and MATCH /ARG2  (:G ::ARGS2)
and  OPPOSITE (F) = G
and  MATCH /ARGS1  /ARGS2
then "TRIVIAL"
```

There are very many rules of this kind which could be included and their complexity can be as great as desired. We could further improve on the rule for OR to make it recognise that

```
(IS-LAND "A") is opposite to
(NOT (IS-SEA (NEIGHBOUR "B")))
```

if "A" is the only neighbour of "B". As the rules become more and more complex they become more like deduction rules. The deduction is only made possible by symbolically manipulating expressions instead of evaluating them.

To show the second way in which we can improve upon our three simple why-functions, we consider (WHYFN NOT). We have assumed that its argument is always of the form (:F ::ARGS). In this case, it is a structured expression whose top-level combinator is APPFN. This is not always so for we could have the question "Why is it not true that if B is on the sea then A is on land but otherwise false is not true?". We could type this in as

```
EVAL (WHYQ(NOT(COND(IS-SEA "B") (ISLAND "A") FALSE)));
```

and the arguments top-level combinator is COND. One reason could be'"B" is on the sea and "A" is not on the land', i.e.

```
(AND (IS-SEA "B") (NOT (IS-LAND "A")))
```

To get at this we could arrange that (WHYFN NOT) evaluates a WHY-NOT-FN attached to the top-level combinator of its argument. In the first case,

this function would be (WHY-NOT-FN APPFN) and would be the simple why-fn we described above for NOT. In the COND case the function would be a (WHY-NOT-FN CONDFN). We can think of it as though we ask the structured expression (NOT expression) to tell us why it is true. It, in turn, asks expression why it is not true and expression behaves differently depending on its top-level combinator. A sequence of questions is asked from expression to sub-expression until a stage is reached where the question can be answered. It is perfectly acceptable that attempts be made to evaluate an answer at any stage of this chain of question answering. For example, NOT might try to give a reason for its sub-expression being false, but fail and have to ask the the sub-expression itself.

This situation is similar to that which faces WHYQ. So far we have only applied WHYQ to expressions of the form (:F ::ARGS). In this case, the top-level combinator of WHYQ's argument is APPFN. We could instead ask

```
EVAL (WHYQ (COND(IS-SEA "A") TRUE FALSE));
```

In this case WHYQ should set the WHYFN of CONDFN to work.

(WHYFN CONDFN) takes three arguments, say A, B and C. It behaves as follows

```
If A evaluates to be true
then LISP (AND /A /B)
else LISP (AND (NOT /A) /C)
```

So far we have supplied all the WHYFNs ourselves although in our sketch we suggested that the why-function of F could perhaps be generated from a definition of F. A simple case of this is

```
     EVAL (WHYQ (IS-LAND "A"));
where IS-LAND has been defined by
     EVAL (SETQ IS-LAND (LAMBDA(X) (NOT(IS-SEA X))));
```

The result should be (NOT (IS-SEA "A")) so let us see how this is attained. First of all WHYQ sees that its argument matches (:F ::ARGS). F is in fact VALUE(%"F"%) which upon evaluation returns the definition of IS-LAND. WHYQ assigns this definition to F, so

```
F = (LAMBDA(X) (NOT(IS-SEA X))).
```

The reason is now easy to generate. WHYQ simply replaces all occurences of free variables in the body of F with the values they take when F is applied to ARGS. In this case X is replaced by "A". We have a function EVALFRQ which replaces free variables by their current values so the result is produced as follows :-

1. Match F with (LAMBDA :PAR :BODY)
2. Produce the expression

   LISP ((LAMBDAQ /PAR (EVALFRQ /BODY)) //ARGS)
3. Evaluate this expression thus evaluating (EVALFRQ /BODY) in the relevant environment.

Notice that in Step 2 we use LAMBDAQ instead of LAMBDA. This means that the reason for (IS-LAND (NEIGHBOUR "B")) is (NOT (IS-SEA(NEIGHBOUR "B"))) rather than (NOT (IS-SEA "A")).

We now have two ways in which the WHYFN of F may be found. Either it is provided by the user or the definition of F is used. In our system both methods may be used for the same F. User provided functions are tried first. Consider as an example the function POSITIVE defined by

```
EVAL (SETQ POSITIVE (LAMBDA(X) (GREATER X 0)));
```

If we say

```
EVAL (WHYQ (POSITIVE 3));
```

then the definition will be used and the result will be

```
(GREATER 3 0).
```

In the special case

```
EVAL (WHYQ (POSITIVE (MINUS 3 2)));
```

a user defined why-function is used which tries to match its argument with

(MINUS :A :B). It returns the result

(GREATER 3 2).

If in this case the definition had been used the result would have been the less satisfactory

(GREATER (MINUS 3 2) 0).

(WHYFN POSITIVE) takes one argument which is a structured expression and it has to give a reason for that expression being positive. It behaves very similarly to the function WHYQ. If its argument expression is of the form (:F ::ARGS) then it gets a function associated with APPFN called (WHY-POS-FN APPFN). This, in turn, can deal with cases where F has a WHY-POS-FN attached to it and also where the definition of F needs to be used. If the argument of (WHYFN POSITIVE) was of the form (COND :A :B :C) then (WHYFN POSITIVE) appeals to (WHY-POS-FN CONDFN). If the argument had been simply a number, then the result would have been "TRIVIAL". Lets take an example of each of these cases.

(WHY-POS-FN DIVIDE) takes two arguments A and B say. It is used in questions of the form

EVAL (WHYQ (POSITIVE (DIVIDE 3 2)));

The result is that both A and B are positive or that both are negative. To determine this, A and B are evaluated. Thus the reason for the example above is the expression

(AND (POSITIVE 3) (POSITIVE 2)).

(WHY-POS-FN CONDFN) takes three arguments A, B and C, say. It evaluates A, and if the result is true, the reason is

LISP (AND /A (POSITIVE /B))

otherwise the reason is

```
LISP (AND (NOT /A) (POSITIVE /C))
```

(WHY-POS-FN LAMBDAFN). This behaves very similarly to (WHY-FN LAMBDAFN). The reason that a lambda expression applied to arguments ARGS is positive is that its body is positive when the parameter variables are bound to the arguments. The procedure followed for a lambda expression F is

1. Match F with (LAMBDA :PAR :BODY)
2. Produce the expression

```
LISP ((LAMDAQ /PAR
      (EVALFRQ (POSITIVE /BODY))) //ARGS);
```

3. Evaluate the expression thus evaluating
   (EVALFRQ (POSITIVE/BODY)) in the relevant environment.

If we define DOUBLE by

```
EVAL (SETQ DOUBLE (LAMBDA(X) (PLUS XX)));
```

then we can type

```
EVAL (WHYQ (POSITIVE(DOUBLE 2)));
```

with the result

```
(POSITIVE (PLUS 2 2)).
```

We could apply WHYQ again to this reason and would get the answer

```
(AND (POSITIVE 2) (POSITIVE 2)).
```

A final example

We will combine all the techniques we have described to give a final example of output from the system. First of all we define "the coastal effect of PLACE1 on the temperature of PLACE2" to be

```
EVAL (SETQ COASTAL-EFFECT (LAMBDA (PLACE1 PLACE2)
   (COND (ISLAND PLACE1) 0
      (DIVIDE (MINUS (TEMPERATURE PLACE1)
                     (TEMPERATURE PLACE2))
              4))));
```

In words, "A place on sea reduces the temperature of a neighbouring place on land by a quarter of the difference between their two temperatures". Given this definition we can ask why the coastal-effect of "B" on "A" is positive. In the printout given below, the function ALL-THE-REASONS-FOR finds a reason and then finds a reason for that, and so on. Each reason is printed out both in a tidy form and in English.

```
EVAL (ALL-THE-REASONS-FOR (POSITIVE (CE "B" "A")));

(POSITIVE                       THE EFFECT OF "B" ON THE TEMPERATURE
   (COASTAL-EFFECT              OF "A" IS POSITIVE.
     "B"
     "A"))

WHICH IS TRUE BECAUSE

(POSITIVE                       IF "B" IS ON LAND THEN THE RESULT IS Ø
   (COND                        WHICH IS POSITIVE, OR ELSE THE RESULT IS
      (ISLAND                   THE DIFFERENCE OF THE TEMPERATURE OF "B"
        "B")                    AND THE TEMPERATURE OF "A" DIVIDED BY 4
      Ø                         AND THAT IS POSITIVE.

   (DIVIDE
     (MINUS
       (TEMPERATURE
         "B")
       (TEMPERATURE
         "A"))

     4)))

WHICH IS TRUE BECAUSE

(AND                            IT IS FALSE THAT "B" IS ON LAND AND THE
  (NOT                          DIFFERENCE OF THE TEMPERATURE OF "B" AND
    (ISLAND                     THE TEMPERATURE OF "A" DIVIDED BY 4 IS
      "B"))                     POSITIVE.
  (POSITIVE
    (DIVIDE
      (MINUS
        (TEMPERATURE
          "B")
        (TEMPERATURE
          "A"))

   4)))

WHICH IS TRUE BECAUSE

(AND                            "B" IS ON THE SEA AND THE DIFFERENCE OF THE
  (ISSEA                        TEMPERATURE OF "B" AND THE TEMPERATURE OF
    "B")                        "A" IS POSITIVE AND 4 IS POSITIVE.
  (AND
    (POSITIVE
      (MINUS
        (TEMPERATURE
          "B")
        (TEMPERATURE
          "A")))
    (POSITIVE
      4)))

WHICH IS TRUE BECAUSE

(GREATER                        THE TEMPERATURE OF "B" IS GREATER THAN
  (TEMPERATURE                  THE TEMPERATURE OF "A".
    "B")
  (TEMPERATURE
    "A"))

AND I DONT KNOW WHY THAT IS TRUE.
:
```

## 6.5 Output functions

The English-like explanations in the last example were produced by the program in an interesting way which is very similar to the way why questions are answered. To make the program capable of answering why questions, we associated various functions (for example why-functions and why-positive functions) with the basic functions of the system. This enabled any structured expression to give an explanation of itself. Similarly, we have associated output-functions to the basic functions and combinators of the system so that any expression, by utilizing its subexpressions, is able to print itself out in English. The output-function for the top-level function of an expression may either examine its subexpressions or ask them to print themselves out. Since we have procedurally embedded the knowledge of generating English, any output function can make use of its environment at the time it is called. In particular, its control environment will contain information about the structure of the textual and semantic context of its own output. In principle, an output-function could examine models of the situation or make any deductions whatsoever.

The output functions we have written are quite simple. We will describe some of them and then give a few examples to show how the power of this method could be used.

The output function of AND prints out its first argument, then prints out "and", and then prints out its second argument. It prints its arguments by asking them to print themselves in English as do the other output functions.

The output function associated with TEMPERATURE prints out "the temperature of" followed by its argument.

MINUS prints out "the difference between" and then prints out AND of its two arguments.

POSITIVE needs to be a little more complex than these as can be seen from the second explanation in our example. It usually prints its argument followed by "is positive" but if the argument is of the form (COND /A /B /C) it needs to behave differently. Then it prints "if" followed by printing

out A, followed by "then", followed by printing (POSITIVE /B), and so on.

In our program NOT printed out using the construction "it is false that". It would be more clever to make NOT set a flag which caused any verb nested within it to change to the negative form.

We could also improve the output-function of AND. It could examine its arguments to see if factoring is possible. Then

(AND (POSITIVE 2) (POSITIVE 3))

would print out

2 and 3 are positive

Similarly, MINUS could factor out its arguments to print "the difference of the temperatures of "A" and "B" instead of "the temperature of A minus the temperature of B".

A particularly interesting problem in this area would be to see how outputs may be punctuated or transformed to remove ambiguity. For example

"The difference of 4 and 8 divided by 2"

is ambiguous as are many of the outputs given in our example explanation.

Our method of programming, because it was intensional, makes it possible to have context sensitivity in producing English output.

## 6.6 Summary

Our program provides illustrations for some of the main points put forward in Chapters 1 and 3. The distribution principle is particularly well illustrated since we have distributed throughout the program and have allocated why-functions, why-positive-functions, output-functions and others to a number of primitive and non-primitive functions of the system. This also illustrates clustering of functions, since any function may have a number of these others attached to it. Since a structured expression can be both

examined and evaluated, and since it may have a number of other possible behaviours, structured expressions are similar to the active descriptions suggested in Chapter 2, although our LISP system has no parallelism. Finally the intensional nature of structured descriptions relates them to ACTORS which we discussed in connection with intensionality in Chapter 3. We will return to this in a moment as the last of three systems which are related to our program.

First of all, there is a relationship between our LISP system and the special LISP interpreter of Boyer and Moore (1972) which was designed for proving theorems about LISP programs. Their interpreter was extended to know facts like "(CAR(CONS A B)) should evaluate to A whatever A and B are". Before evaluating an expression it checked to see if that expression was in one of the forms it could recognise for direct symbolic interpretation. The difference between the two systems is that in the Boyer-Moore system the evaluator itself checks for the various possibilities whereas in our system the checks are associated with a relevant function either primitive or user defined. Our system is designed so that it is natural for the user to define functions with symbolic evaluation knowledge. We could, for instance, express the knowledge about CAR stated above as

```
FUNCTION CAR1 X;
DECLARE LOCALS A AND B;
IF MATCH /X (CONS :A :B)
THEN A
ELSE CAR (X)
CLOSE
END;
```

We suggest that it is best to distribute such idiosyncratic knowledge of evaluation so that it can be put in appropriate places. As more and more cases are included, distribution becomes more efficient than centralisation because there is no need to search through all possible pieces of expertise. It is also clearer to understand and modify a distributed program.

There is also a clear link between the idea of passing all arguments unevaluated and pattern matching invocation in PLANNER and CONNIVER type languages. For instance, GREATER could be written in POPLER as

```
PROCEDURE INFER <<GREATER £>X £>Y>>;
procedure body ...
ENDPROC;
```

and X and Y might get instantiated to be patterns. If we said for example

```
GOAL <<GREATER <<TEMPERATURE A>> <<TEMPERATURE B>> >>;
```

then X would be bound to <<TEMPERATURE A>>. X and Y could be compared without being evaluated. Again we see the ability to examine the structured description of a goal in the form of a pattern in this case is what allows a program to do symbolic reasoning.

We believe that execution of programs is a generalisation of deduction, and although we cannot prove this statement we can find many persuasive analogies between the two processes. In mathematical logic there is a world about which certain statements are true, and proof procedures are the way of finding out which. If we concentrate on proofs we find that the look like programs. There are program steps like apply axiom 4 to statements 73 and 54, and these have resulting side-effects which add new statements to the proof. Lemmas and theorems are a clear analogy to subroutines. McCarthy (1970) has pointed out the relationship between recursion and induction, and Moore (1973) has made it central to his program for proving theorems about LISP programs. On the other hand, a process need not always mirror a particular logical deduction : consider the growth of a tree for example. The analogy has implications for our representations of human reasoning which very often appears illogical though appropriate.

The third system to which our program is related is the ACTOR system. As we said in Chapter 3, one view is that an ACTOR is something which can be given a variety of messages and which has an intensionally defined behaviour. In particular, one useful way of specifying the behaviour of an ACTOR is to treat the various cases of message separately and consider what needs to happen for each. This would be a good way of associating evaluation-functions, why-functions, and so on, with the structured expressions of our system. In addition, with the powerful and expressive heterarchical

control facilities available in ACTOR systems we could extend our program to do more complex deduction and could attempt to tackle the control problems involved in producing relevant explanations and explanations by dialogue. This seems a most promising line of research.

# PART II. Control structure

## Introduction

In part I we discussed a theory of knowledge representation which is based on our distribution principle. The first three chapters reviewed the declarative / procedural problem and developed a method of representation which grouped procedures into clusters about particular themes. Chapters 4 to 6 took three different domains and described a program from each. The programs illustrated various aspects of our theoretical study. At certain points, for example in section 6. 4, we highlighted problems concerned with organising the behaviour of these programs. Representing knowledge as a collection of operationally defined units meant that they must invoke one another to interact. This is different from the situation where knowledge is expressed as data for a set of uniform retrieval procedures. It leads to deep nesting of function calls and is restrictive if simple control structures are used. Sometimes we might really want one unit to call another without forcing him to return. Sometimes a unit might only temporarily return control. These facilities lead to complex control problems and we investigate them thoroughly in this part of the thesis. In this introduction we make some definitions and describe our presentation but first of all we set the scene for our work.

Artificial intelligence has benefited from high-level computing languages in very many ways, and some high-level languages have even been developed specially for uses in this field. There have been notable exceptions of course. For instance, Samuel's chequers program was a large AI program written in machine code. However, the structures provided by recursive languages like POP-2, LISP and ALGOL-60 have contributed much to the structuring of AI programs and perhaps even more towards the structuring of AI thinking and philosophy. Sussman and Mcdermott make the point (1972) that "A higher level language derives its greater power from the fact that it tends to impose structure on the problem solving behaviour of the user". Over the last ten years there has been an increase in the number of control structures provided in programming languages. Applications have come to light which have led to the use of control ideas such as back-

tracking, saving states, coroutining, generalised jumps and so on. We have developed a system called P-74 which generalises these structures, unifying them and suggesting new ones.

A run-time structure of a program is any structure produced explicitly when a program is executed but which was only implicit in the static text of the program. In particular, run-time structures have two important properties: they change dynamically as the program progresses, and they vary from one execution of a program to another depending on the input and initial conditions.

A very important facility provided by most high-level languages is the ability to structure programs more expressively than by flow-charting techniques alone. All building blocks in a flowchart are instructions. They are tied together by their consecutiveness, by conditionals, and by jump-to-a-pre-defined-label instructions. A high-level language, however, allows any flowchart of instructions to be considered as a whole and used as a single instruction. In other words, the programmer, and even the program, is allowed to define useful larger building blocks of text to be interpreted by the machine. In POP-2 we call these functions, in ALGOL they are termed procedures, in LISP they are progs or functions and in machine code they are sub-routines. The general, language independent term for them is "module".

Modules are represented by textual definitions (e. g. block definitions) which are textually nested. A module may introduce new nomenclature, call other modules, and sequence instructions within itself by means of flowcharting. When a program in one of these languages is executed, any module which is active may pass over control to some other previously inactive module provided it can refer to that module. This gives rise to a dynamic pattern of activity. Since module activations may set up links in order to remember who passed control to them, it also gives rise to structures called run-time control structures which represent some of this past activity. These structures which govern the flow of control of programs and the accessibility of variables are the structures we will consider.

Programming languages provide various automatic ways in which flow of control can be dynamically organised. Each of these arrangements, or regimes, has possible variations. We will examine some regimes to make the range of possibilities more clear, and will introduce a regime of

our own called P-74. P-74 incorporates a new concept of layers of control structure which clarifies the relationships and suggests new ones. Relationships between modules form the first layer of control structure and allow us to write one-layer programs. Relationships between one-layer programs form the second layer, and so on.

We survey three relationships; the subroutine relationship, the semicoroutine relationship, and the coroutine relationship. We take each relationship in turn and describe the primitives for it. In doing so we introduce "activation records". These are used to implement the primitives, and provide a clear model for explaining them. Since they were developed, the three relationships have appeared in various forms. In particular they have been used to relate modules and also to relate programs which themselves have one layer. This distinction is important and the two options correspond to the innermost two layers in our system of layers. Relations between modules correspond to the innermost layer and relations between programs to the next innermost. We describe the two options separately in the first two chapters of this part.

In chapter 7 we define the primitives for the innermost layer of P-74 and use these to give examples of all the relationships, including some given elsewhere in the literature. The three relationships can be combined in various ways to make differing control regimes. We recognise four regimes namely subroutine, coroutine, semi-coroutine, and full coroutine regimes. The subroutine regime is also called a dendrarchy by Bobrow and Wegbreit, who proposed a system to unify the various regimes (1972, 1973). They used a primitive called ENVEVAL. We demonstrate an interesting relationship between their scheme and our system of layers. In particular, by modifying ENVEVAL we can immediately and very naturally express all the three relationships in their various forms.

Having discussed relationships between modules, the way is open for a discussion of the other layers provided in P-74. Chapter 8 treats the relationships again, but between programs which already have one layer of control structure. It introduces two-layered programs. The language SIMULA provides these, giving the user a full-coroutine regime between one-layer programs. We describe the primitives of SIMULA and compare them with their counterparts in P-74. The primitives of P-74 at layer one are almost identical with those at layer two and P-74 generalises

systems like SIMULA which have different primitives at each layer.

After summarising primitives for the first two layers of P-74, we give some example programs which use them. The next sections survey jumpout, back-tracking, and generalised jumps. We show how our primitives clarify these. Finally, we show how to generalise our modification of ENVEVAL to two layers. This allows us to express very concisely all the primitives so far.

In chapter 9 we consider programs with more than two layers, and give both an intuitive and a more formal specification of P-74's general primitives. We give some examples using these. Other questions concerning control structure naturally arise and we answer these by comparing P-74 with the ACTOR system of Hewitt (1973).

Although P-74 is a logical extension and generalisation of other control structures, our conclusion is that all layers of it except the first two are cumbersome to use. This leads us to argue that control structures should be considered as patterns of message passing (Hewitt 1973). The activation record approach should only be used in particularly regular situations.

# Chapter 7. Relationships between modules

## 7.1. The subroutine relationship and activation records

In this chapter we describe the subroutine, semi-coroutine and coroutine relationships as applied to modules of code. A module of code can be thought of as a flowchart composed of primitive instructions and conditional and unconditional jumps. We generally represent a module pictorially as a box, as in figure 30, and we may draw a schematic flowchart in the box. A module is like a function in POP-2 or LISP, and like a procedure in ALGOL.

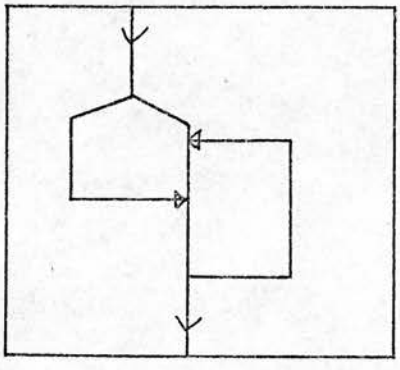


Figure 30. A module.

If a module, say F, makes no calls on any other module, whenever it is executed control will simply flow from its entrance to its exit along some path of its flowchart. The program counter, which always points to the next instruction to be executed, will anticipate control along this path. If, however, F makes a subroutine call to G while only partway through its own code, control will jump to the beginning of G's flowchart. In the subroutine relationship, G is primed to return control to F which has only been temporarily halted. Figure 31 shows the flow of control in this case. G is said to be a subroutine of F. We say that F calls G and that G returns to F. We will introduce corresponding terminology for the other relationships later. Call corresponds to procedure entry in ALGOL-60, and return corresponds to procedure exit.

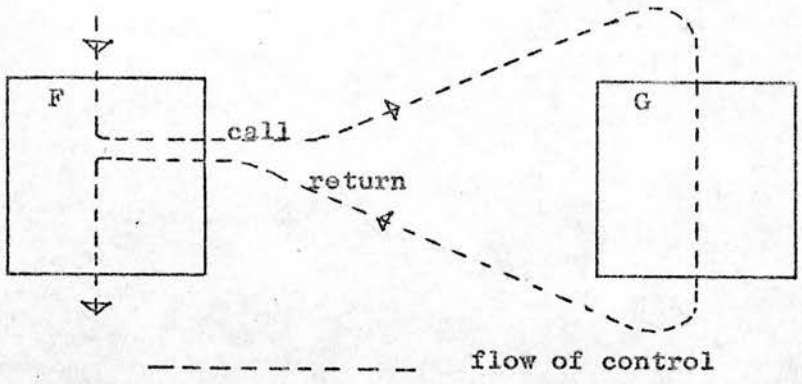



Figure 31. Call and return.

Since G must return control to F, something must store the point in F to which control is returned. There are three standard ways of doing this.

1) When F calls G it tells G where to return to. G stores the return location pointer in a special cell within itself and returns control to the correct place in F by way of this pointer. This corresponds to the "JMS" instruction in the PDP-7 machine, which stores the program counter one location before the instruction it jumps to. G must remember to return to F, and also where in F to return to.

2) F remembers where it is to be continued by storing the return location in a special location within itself. G now returns by way of this special location. G needs only to store the address of the special location in F and not information about where to continue F.

3) F stores the return location in a special place neither in F nor in G. G need only remember to return to the address in the special place.

Notice that there are two separate decisions; which module to return to, and where in that module to continue.

Some of these methods are discussed in Knuth (1968), and in Wegner (1971). Sometimes other values, such as values of variables or of registers, need to be saved too. These can be stored in any of the ways mentioned.

If we allow subroutines to call subroutines themselves then we get an interesting pattern of behaviour. At any time there is a chain of links back from the current subroutine through those which called it. This chain of links grows whenever a subroutine is called and diminishes when a subroutine returns. Since calls and returns are dynamically nested in the sense that a return always corresponds to the latest preceding call, the chain can only change at its active end and so behaves as a "last in first out"(LIFO) stack. The stack traces out a tree through time; this is why the regime is called a dendrarchy.

The flow of control is more implicit in a dendrarchy than it is in a flowchart program where points for control to jump to must be specified absolutely. The instruction "call subroutine" means "go away and do the subroutine but remember to come back here when you've finished". The subroutine will contain a return instruction meaning "go back to whoever called you". This is clearly an implicit jump instruction since the location depends on where control has been up to that time rather than on the structure of the program at the time of programming. It is a dynamic, process-oriented decision rather than a static, textually-oriented one.

Figure 32 shows the situation when F has called G, G has returned, F has called H, and H has called K.

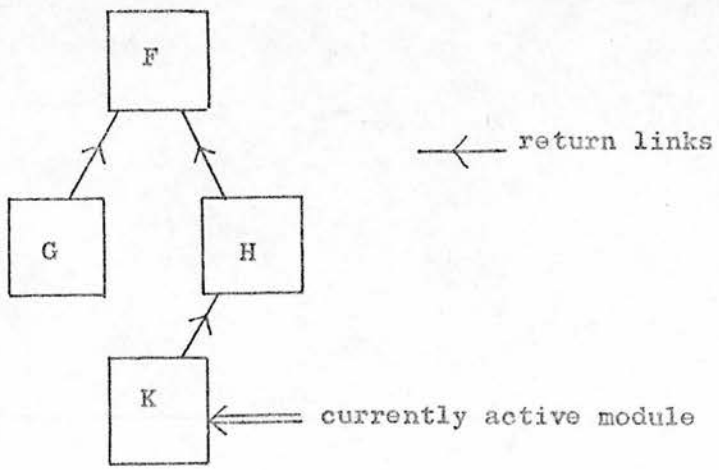



Figure 32.

Let us examine how each of the three ways of implementation cope with this situation. Case 1) manages since H remembers to return to F, and

K remembers to return to H. They each know where to continue in their respective return modules. G, by chance, still knows where it had returned to in F. In case 2) the modules H and K know where they should be restarted. G, by chance, remembers that it had returned to F but the special location in F has been overwritten by the continuation point for H. Case 3) does not work at all. The special place reserved for continuation points only retains the last return location, namely the one in H. K can successfully return to H but H then tries to return to itself.

Thus only cases 1 and 2 will operate under the extension to a dendrarchy. If we now allow the dendrarchy to be recursive then none of the cases will work. The dendrarchy becomes recursive when two activations of some module F are nested. This happens if F is called while another activation of F is on the current return chain. Figure 33 illustrates the problem.

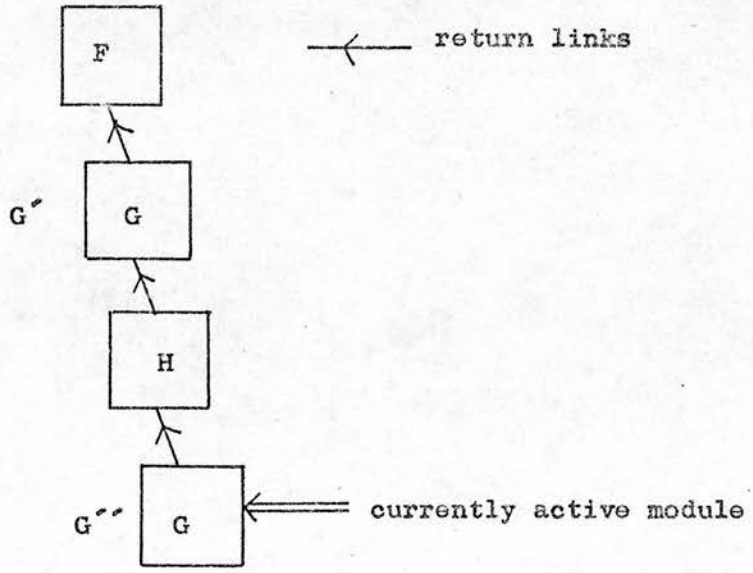



Figure 33. A module called recursively.

Here, there are two instances of G being activated, namely G` and G``. G` must remember to return to F, and G`` must remember to return to H. But there is only one location in G for storing this information. In case 1 it stores the place in H that control must return to, and in case 2 it stores only the fact that H is the module to be returned to. The problem is that there can be any number of current activations of a module but only one object represents them all. A module becomes confused when it has to remember several return modules, one for each

activation of the module. We can solve the problem by having an object to represent each activation. This is usually called an activation record. Each has a pointer to the record representing the module activation which called it. The pointers are called return activation links and they link activation records into a return chain. A new activation record is set up at each subroutine call. All information associated with that particular call, including its return activation link, is stored in the record. The code for the module then never needs changing and becomes re-entrant.

Dijkstra noticed (1960) that in the recursive subroutine situation the run-time structure of activation records behaves like a LIFO stack. The third method given above for storing return locations can thus be made to work by replacing its special storage location with a special stack which stores the chain of activation records. An activation record then no longer needs an explicit return pointer since its return activation record is just the previous record on the stack. This method can be modified to work for the other two cases but less satisfactorily.

Although a LIFO stack is an efficient representation for a subroutine regime, it will not easily cope with more general regimes. Since we might wish to re-enter an exited function, we must keep information about that function somewhere. If it is stored on the stack it is likely to be overwritten when the stack grows. Besides this, the user and his program are not generally allowed to access the stack. It is supposed to work behind the scenes so that it can be coded efficiently, but in a compressed form intelligible only to the system. Bobrow and Wegbreit (1972) implement a regime which we will discuss in section 7.4, and although they use frames, which are similar to activation records, they have devised an efficient way of organising them. In simple situations this behaves as a stack and in more complex ones as a dynamic storage allocation scheme with a garbage collection for frames.

Activation records and frames give us two advantages over usual stack implementations. Firstly, they are easily manipulable objects which represent function activations. Althought this is less efficient for simple recursive situations it allows us to deal with more complex ones. We can save a pointer to a context in the current control chain for use even after we have exited from it. Since we have a pointer to

its activation record, this and its ancestors will not be garbaged when we exit. Moreover, since the dendrarchy is grown using new storage, further function entries will not overwrite needed activations.

The use of activation records is illustrated in figure 34. This shows the situation when F calls G which calls H which again calls G which again calls H. G and H each have two corresponding activation records. The return chain from the current activation record H2 is five activations long.

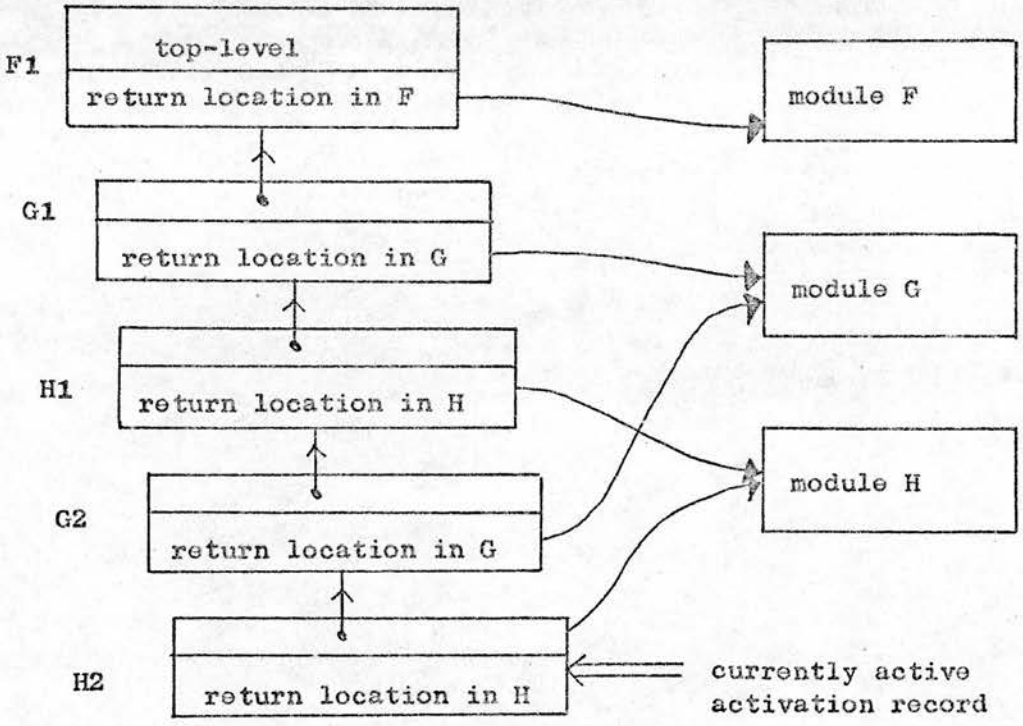



Figure 34. Activation records used to implement recursion.

Recalling the distinction between a module remembering which module to return to, and remembering where to return to in that module, we see that there are two places in which we could store return locations in an activation record implementation.

1) We can store the return location of an activation of a module in its corresponding activation record.
2) We can store the return location of an activation of a module in the activation record of any module which it called as a subroutine.

Bobrow and Wegbreit point out that the first of these is the more natural. If the subroutine relationship is used alone then the difference is negligible, but with other relationships several module activations may be capable of returning to any one activation. In this case, with the second method, the activation would become confused. We shall see an instance of this case with the semi-coroutine relationship.

## 7.2 The semi-coroutine relationship

The semi-coroutine relationship has only been referred to in the literature as a relationship between entire recursive programs, but for completeness we shall deal with it first as a relationship between modules (Dahl & Hoare, 1972). It is a direct extension of the subroutine relationship.

There are two asymmetries in the calling situation of subroutines. Firstly, the caller, whom we name the master (after Dahl), is activated both before calling a subroutine and again after the subroutine has returned to it. The subroutine effectively continues execution of its caller but cannot be recontinued by it. Secondly, the master has a choice of which servants he runs whereas the servants are obliged to return only to their master, although of course they can have servants themselves.

By relaxing the first of these constraints, so that a master may continue his servants after they have returned to him, we have a semi-coroutine relationship between master and servant. The master and servant can, metaphorically at least, hold a discussion. This situation can be useful if, for example, a master wishes his servant to produce different results, one at a time. Defining some terminology, we say that in this relationship the master runs the servant and the servant rises to his master.

How does a master know how to continue his servant. We wish the activation record for the servant to be made available to the master but we do not wish to prime the master to return to his servant. Two methods can be used to achieve this.

1) If a master wishes to run a module he must first initialise the module by using the procedure "NEW" which returns an activation record as value. He may then run the activation record and on return will find that it has been updated to the latest continuation point for the procedure. Thus he can continue the activation record by running it again. This is the method used in SIMULA (Dahl & Nygaard, 1966).

2) When the servant returns he may store his own activation record in a special location called "CONTINUEE", and the master can use this to access and continue the servant. This method is automatic in P-74 though the first method is also possible.

There is very little difference between these methods in the semi-coroutine situation since a master always knows who he has run. However, in a coroutine situation, as we shall see, an activation may have control passed to it from another which it did not previously know about. The activation receiving control will not know who has continued it unless the second method is used.

Figure 35. illustrates a semi-coroutine situation where F runs G who rises to F, who runs G again. The activation record for F is used three times at this point and that for G is used twice. The code for this example is as follows.

```
FUNCTION F;
"GACT" IS A LOCAL VARIABLE;
"I" IS A LOCAL VARIABLE;
0->N;
STORE A NEW INSTANCE OF
 G IN GACT;
LOOP:PRINT(N);
  2+N->N;RUN(GACT);
  GOTO LOOP
END;

:F();
0 1 2 3 4
INTERRUPT:
```

```
FUNCTION G;
"N" IS A LOCAL VARIABLE;
1->N;
LOOP:PRINT(N);
  2+N->N;RISE();
  GOTO LOOP
END;
```

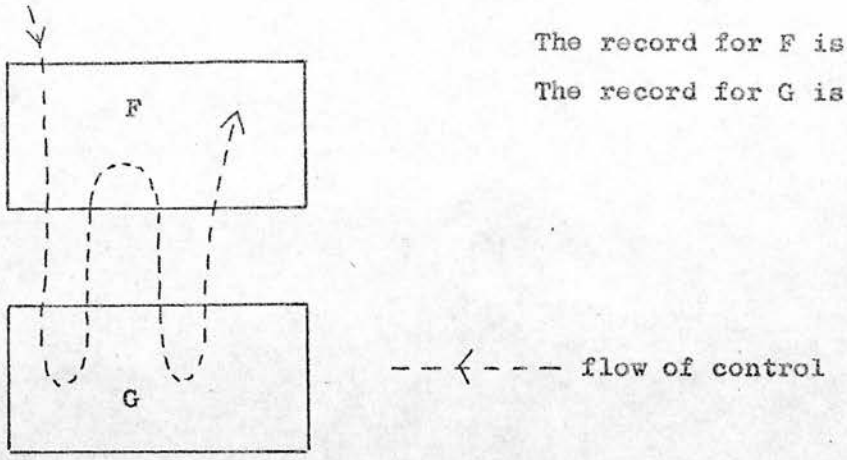



Figure 35. The semi-coroutine relationship.

Since activation records change state when they are run they can be thought of as representing snapshots of a process. We define a process to be any ongoing activity which can receive messages and send messages at any time during its activation. The execution of F in figure 35 is split into three active phases, and at the end of each the activation represents the instantaneous state of F. It has a program counter which is advanced every time the record is continued by a run or rise. Moreover, activation records can be passed about as arguments, results, or values of variables, since they are objects in their own right. We have already seen an instance of this at a rise, when the servant activation is assigned to the variable "CONTINUEE". This is a property which subroutine regimes lack. In subroutine regimes activation records are usually inaccessible.

The changing internal state of activation records is an important property. If a particular record R is stored in variables A & B, and is run from A, then, when control returns to the master, both A and B will have been altered. Running R from A side-effects B. Such side-effects imply that if two processes know a third, it can be continued from either. We can avoid side-effects if we arrange that an activation record is copied before it is run, and that the copy is run in its place. We call regimes which make copies "with-copy" regimes: the regime we have just discussed is the semi-coroutine regime without copy. We will return to this distinction later when we consider ACTOR-like systems.

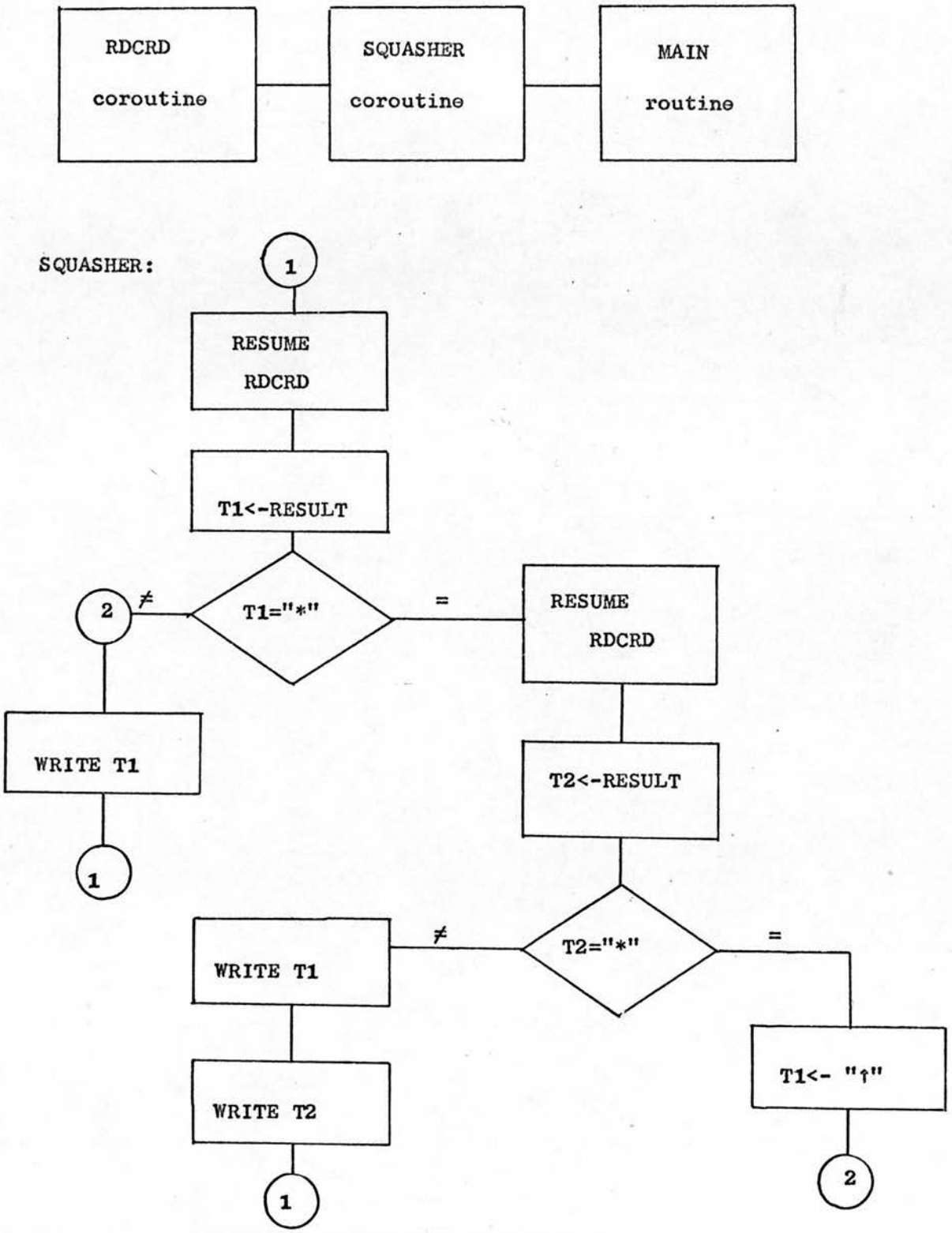
## 7.3 The coroutine relationship

The coroutine relationship was discovered by Conway and Erdwinn (1963). It is discussed in Knuth (1968) and in Wegner (1971) as a relationship between modules, and in Dahl and Nygaard (1966) as a relationship between ALGOL programs. In this section we shall deal with the coroutine relationship between modules.

When two activations are in a coroutine relationship there is complete symmetry between them. Each one may pass control to the other but leaves it under no obligation to return. Control may pass freely within a system of coroutines provided each has access to any it wishes to continue. If one coroutine passes control to another it is said to resume it.

Conway gives a simple example of a situation where coroutines are useful. Suppose we have an input file and an output file. We require a program which reads characters from the input file on cards of 80 characters. It outputs the characters to a main program. The program is also required to replace any pair of asterisks "**" by a slash "/", a replacement which occurs in FORTRAN and COBOL compilers. A simple way to code this algorithm is as three separate routines. One reads cards, one replaces "**"'s and the main program uses the output from the second. The three routines are called "RDCRD"(read card), "SQUASHER"(compress asterisks), and "MAIN-ROUTINE". They are shown schematically in figure 36, which also shows the flowchart for "SQUASHER". Each routine asks its previous one for characters one at a time, as it needs them. The main program asks "SQUASHER" for its next character and "SQUASHER" asks "RDCRD" either once or twice, according to the character "RDCRD" gives it. If "RDCRD" gives it an asterisk then "SQUASHER" must check if the next character is also an asterisk. The routines must communicate as coroutines since each must remember where it is when it returns control to the others.

We call this regime a coroutine regime. It is clear that in such a regime it is advantageous to use the second continuation scheme outlined above, since it is quite common for an activation to need to know who continued it. We provide this continuation scheme automatically in P-74.




comments: (1) RESUME RDCRD returns from RDCRD with a character as result.
(2) WRITE T1 resumes the main program with T1 as argument.

Figure 36 (From Conway)

In SIMULA the coroutine regime and the semi-coroutine regime are combined and we name the result a full coroutine regime. In SIMULA 67, the coroutine and semi-coroutine relationships are between programs rather than between modules. The only relationship between modules within these programs is the subroutine relationship, so modules form dendrarchies. We know of no examples of a full coroutine regime between modules except in P-74. This includes full coroutines at every one of its layers. We will discuss the full coroutine regime here using the terminology of SIMULA as though its regime were between modules.

In a full coroutine regime, an activation A may be master of an activation B, and B might resume an activation C as in figure 37.

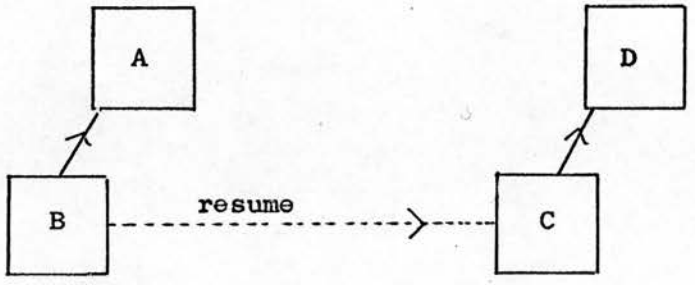



Figure 37. The situation before B resumes C.

The return link of C might point to an activation D, so we must decide what will happen to this link when B resumes C. Since B is a servant of A and is on the same status as C, one natural convention is that C become a servant of A. This is the convention used in SIMULA 67 and we copy it in one version of P-74. Figure 38 shows the new situation after the resume.

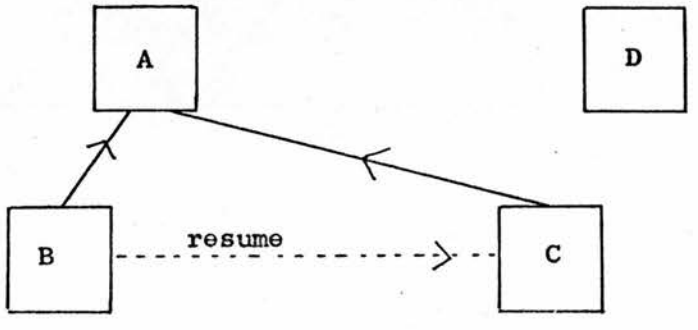



Figure 38. The situation after B resumes C.

This is an instance of a very important problem which we recognise. We feel that the convention was forced upon us because C was incapable of remembering two return activations. We say that C has become confused. This situation corresponds to the one in figure 33, where the module G was required to remember two return modules. In that case the module G became confused. The problem still arises in SIMULA in spite of this convention. Suppose that an activation of F runs an activation of G and that this activation of G runs itself. This is shown in fig 39.

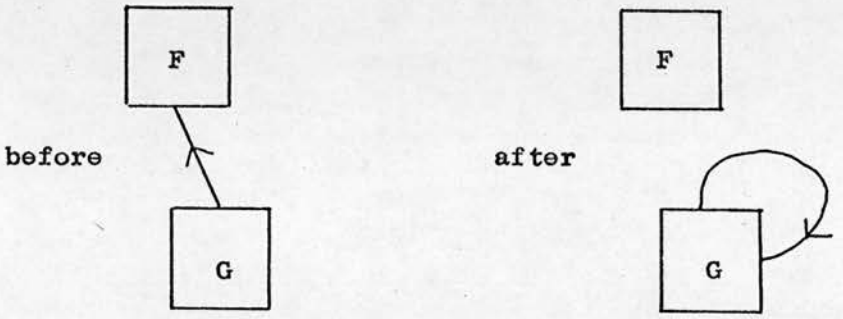



Figure 39. An inescapable loop.

G has got itself into a loop from which there is no possible escape instruction. In P-74 we could escape by doing a higher layer exit. The arbitrariness of this situation deserves discussion and we will return to the problem when we discuss actor-like systems since it provides arguments in favour of ACTORS.

A semi-coroutine regime can simulate a full coroutine regime. Suppose a master M wishes to have a team of servants A, B, C, D,... and to allow these to pass control amongst themselves freely. He has to provide them with a primitive RESUME. If he makes RESUME (X) pass back a message to M using RISE, and if the message says ["RESUME" X] then M can direct control to X. M can arrange that he runs his first servant with a special primitive CORUN, say, which can interpret any message returned from below. If the message is in the above form then CORUN runs the relevant coroutine, otherwise he passes control back up to M by executing RISE.

## 7.4 The enveval regime and P-74

To complete this chapter on relations between activations of modules we will describe a regime due to Bobrow and Wegbreit. We refer to it as an enveval regime after one of its important primitives and will give a variation of it which extends it for incorporation into P-74. In subsequent chapters we will introduce P-74's two-layered and multi-layered version.

Bobrow and Wegbreit give both a model and a stack implementation for their regime. The implementation is efficient and is fully discussed in their paper. The model uses frames, which are more efficient than activation records but also more complex. We now summarise their model relating it to our activation record one. Then, after describing the Bobrow-Wegbreit primitives, we will explain the changes made to them in P-74.

When a module is executed it requires storage for local variables, for parameters, for a return pointer, for an access pointer, and for temporary storage. Each time it is discontinued for any reason, it must also store its own continuation point. In the Bobrow-Wegbreit system, storage is allocated for all this information in a frame for the module. A frame corresponds to an activation record which also stores all information relating to an instance of a module. The difference is that, for efficiency, frames are made up of two parts. These are the basic frame and a frame extension. The basic frame stores information which is known on first entry to the module, in particular the parameters and local variables: its size is fixed on entry to the module. The frame extension contains space for temporary storage used by the module during execution: its size varies during execution. The frame extension has a pointer to the basic frame and is used to store the continuation point. It also has two links called the control link and the access link which point to other frames. Figure 40, taken from Bobrow and Wegbreit, illustrates a frame. Many of the items in the frame illustrated are not discussed here and the reader is referred to the paper.

Control links and access links form two chains of frames called the control chain and the access chain. The control link of a frame

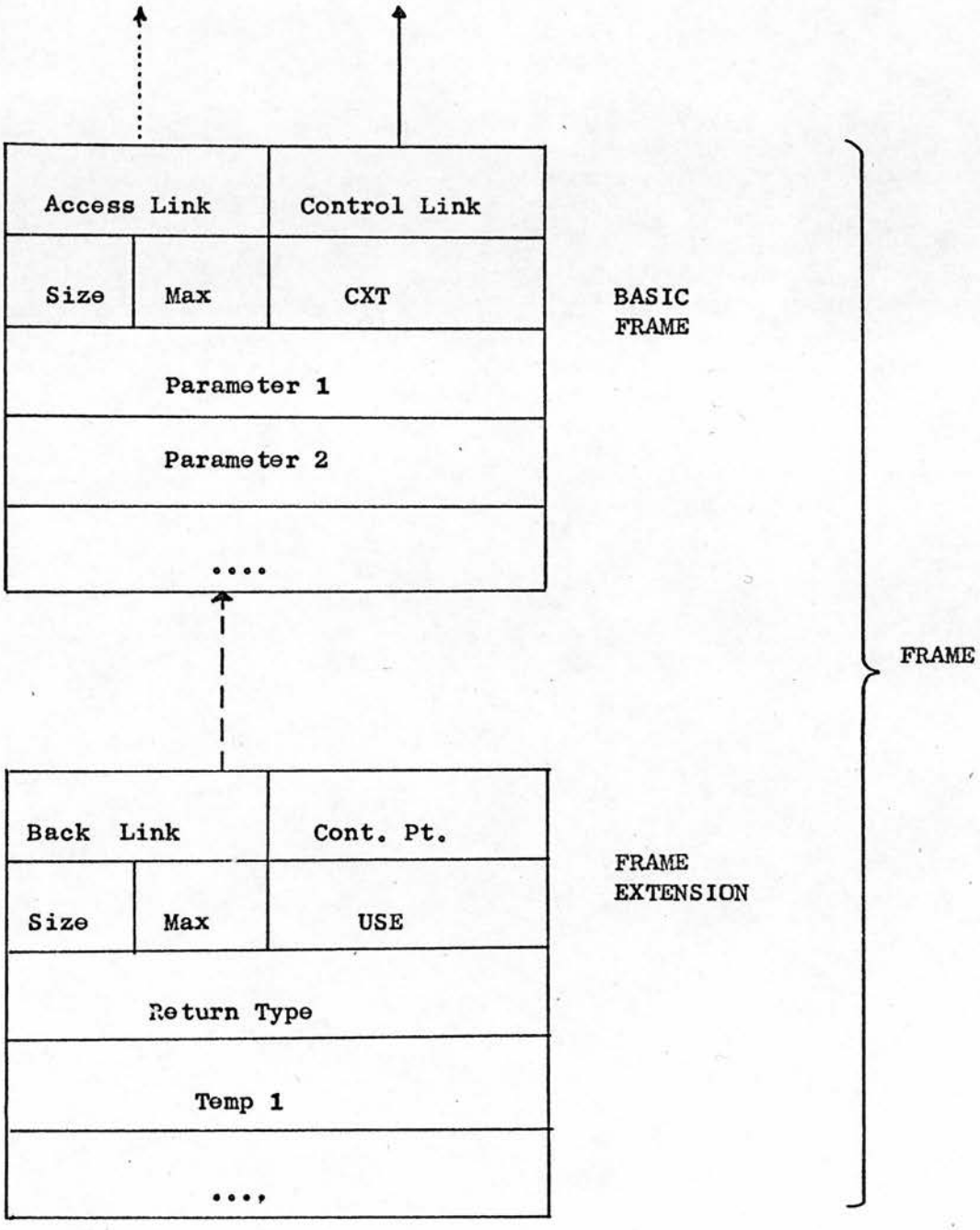



Figure 40. (From Bobrow & Wegbreit)

references the frame to which its corresponding module will usually return control when it finishes. This is the ALGOL "dynamic link" and usually points to the caller's frame. The access chain is the chain of frames used to seek the value of a free variable of the module. In ALGOL, the access link is called the "static link". In POP-2 and LISP it is usually the same as the control link.

In both ALGOL and POP-2, access links and control links are manipulated by the system alone. Control and access environments are regulated automatically and in a fixed way. One of the innovations that the Bobrow-Wegbreit system introduces is that control and access environments can be handled independently. The user can construct a frame from the access environment of one frame and the control environment of another. Moreover, a LISP form (or equivalently an ALGOL expression) may be executed in any frame which is accessible to the user. To allow this, the user must be given access to the control structure, but as the system is implemented in an efficient manner he must not be allowed to access frames directly since he might produce ill-formed ones. He can manipulate them only by means of protected objects called environment descriptors (ed's). Only a few primitives will work on these objects.

The following extract from Bobrow and Wegbreit (1972, pp 10-11) summarises the major primitives that handle frames. The arguments taken by many of these are frame specifications and the forms of frame specification are also given in the extract.

" environ(pos) -- creates an environment descriptor for the frame specified by pos.

setenv (olded,pos) -- changes the contents of an existing environment descriptor olded to point to the frame specified by pos. As a side effect, it releases storage referenced only through previous contents of olded.

mkframe(epos,apos,cpos,bpos,bcopflg) -- creates a new frame and returns an ed for that frame. The frame extension is copied from the frame specified by epos, and the ALINK and CLINK are specified by apos and cpos, respectively. The BLINK points to the basic frame if bcopflg = TRUE. In use, arguments may be omitted; bcopflg is defaulted to FALSE; apos, bpos and cpos are defaulted to epos. Thus mkframe (epos) creates a new frame extension identical to that specified by epos.

enveval(form,apos,cpos) -- creates a new frame and initiates a computation with this environment structure. ALINK and CLINK point to frames specified by apos and cpos, respectively; and form specifies the code to be executed, or the expression to be evaluated in this new environment. If apos or cpos are omitted, they are defaulted to the ALINK or CLINK of this invocation of enveval. Thus, enveval (form) is the usual call to an interpreter, and has the same effect as if the value of form had appeared in place of the simple call to enveval.

A frame specification (i. e., pos, apos, bpos, cpos, epos above) is one of the following:

1) An integer N:
   a. N = 0 specifies the frame allocated on activation of the function environ, setenv, or enveval. In each case, the continuation point is set up so that a value returned to this frame (using enveval) is returned as a value of the original call to environ, setenv or enveval.
   b. N > 0 specifies the frame N links down the control link chain from the N = 0 frame.
   c. N < 0 specifies the frame -N links down the access link chain from the N = 0 frame.

2) A list of two elements (F, N) where F is a framename and N is an integer. This gives the Nth frame with name F, where a positive (negative) value for N specifies the control (access) chain environment.

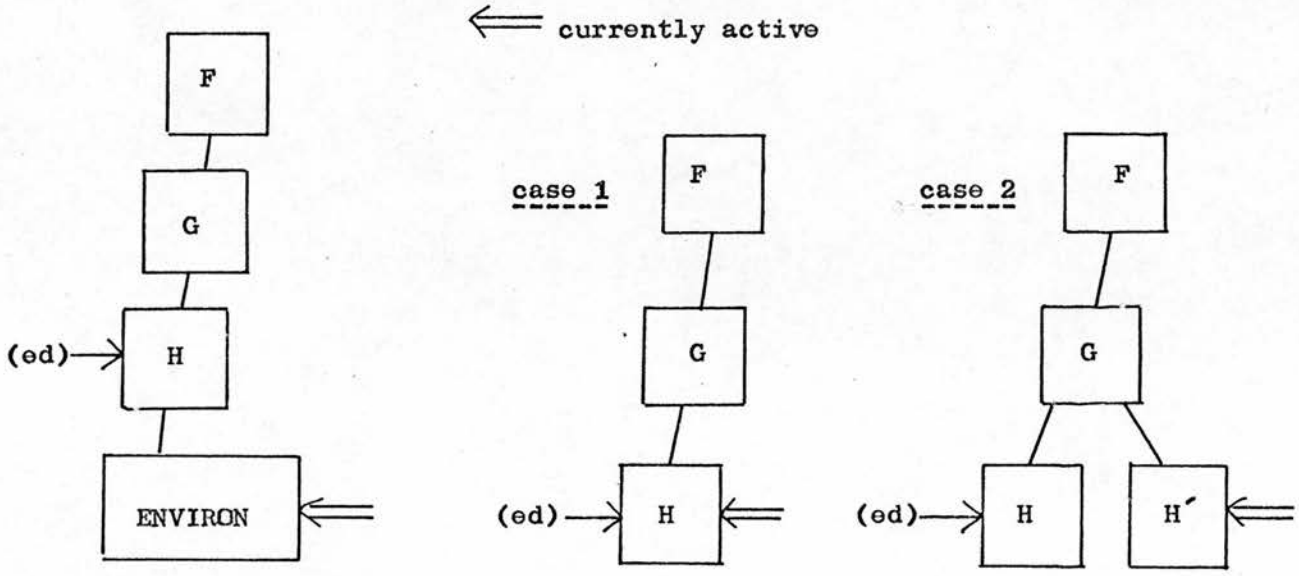
3) The distinguished constant NIL. This value specifies global-access-only to be shared, and / or control-return to the system (process halt). Doing a setenv (ed, NIL) releases frame storage formerly referenced only through ed, without tying up any new storage.

4) An ed (environment descriptor). When given an ed argument created by a prior call on environ, environ creates a new descriptor with the same contents as ed; setenv copies the contents of ed into olded.

5) A list "(ed)" consisting of exactly one ed. The contents of the listed ed are used identically to that of an unlisted ed. However, after this value is used in any of the three functions, setenv (ed, NIL) is done, thus releasing the frame storage formerly referenced only through ed. This has been combined into an argument form rather than allowing the user to do a setenv explicitly because in the call to enveval the contents are needed, so it cannot be done before the call; it cannot be done explicitly after the enveval since control might never return to that point. "

We now return to using our simple activation record model but this will make no major difference to the argument.

Suppose a module F calls a module G which calls a module H. If, while H is active, it makes a call of ENVIRON with argument 1, ENVIRON will return the activation record corresponding to H. The situation is shown in figure 41.




before ENVIRON returns

after ENVIRON returns

Figure 41.

Control will return to H and there are then two possibilities. Either the continuing execution of H uses the same activation or else it uses a copy. In other words, either we have side-effecting of activations so that they represent processes, or else we do not and they represent frozen copies of processes. Let us call the first option an enveval-

without-copy regime and the second an enveval-with-copy regime. We discussed these two possibilities in the case of coroutine, semi-coroutine and full-coroutine regimes.

In an enveval-without-copy regime we must decide whether an activation record represents its immediate control structure (i. e. merely the continuation point within its associated module), or the control structure of its entire control chain. In an enveval-without-copy regime it can represent both at the same time. Suppose, however, we had a with-copy regime, and consider the example in figure 41. Although environ returns to H, it uses a copy so the ed it returns remains unchanged and represents the frozen state of H. There is a problem in using this ed to represent the entire control structure of H, for both H and its copy refer to the same activation of G. This will be modified when H returns control, and the saved environment descriptor will be side-effected. Thus it does not represent the copied return chain. One way to make a with-copy regime do this is for it to use a copy of any activation that control returns to.

Bobrow and Wegbreit's system is such a with-copy one and it very cleverly arranges only to copy when absolutely necessary. This is when the activation being continued is referred to from more than one place, for example along some chain of records from an ed. In Davies' implementation of POPLER 1. 5 which extends the enveval regime to allow more complex backtracking automatically, copying is done in the same way. In P-74 we make the user himself do any copying he requires, and we give him primitives for this. Since P-74 has a system of layers, it makes it much clearer how much of an activation records' control structure a copy of it represents. One-layer activation records represent their immediate control structure only, while two-layer ones represent their entire layer-one return chain.

ENVEVAL is a powerful primitive. Although the Bobrow-Wegbreit model is a one layer model, in that activation records correspond to modules, the freedom of ENVEVAL can be used to simulate two-layer control structures such as backtrack and coroutining between one-layer processes. The system of separating control and access environments allows the funarg problem (Moses, 1970) to be dealt with very nicely and allows frames to have the same access environment whilst having different control environments.

As it stands, however, ENVEVAL cannot be used to simulate a coroutine or semi-coroutine relationship between modules of just one layer. The problem is that only forms (statements) can be used as first argument to enveval, so only forms can be evaluated in arbitrary control environments. In P-74 we extend ENVEVAL to take either a form or an activation record as its argument. In the second case, the activation record is continued in the specified control environment. When it returns, it will pass control to the first record of that environment. At present, however, we have not allowed separation of control and access environments.

Extended ENVEVAL gives us a regime which we call an extended enveval regime. P-74 has a without-copy version. The extended regime immediately gives us a very natural way to implement RUN, RISE, and RESUME as one-layer relationships (i. e. between activations of modules) and so provides a full coroutine regime between modules.

We have:

```
RUN(X)    == ENVEVAL(X  ,ENVIRON(1))
RISE()    == ENVEVAL(NIL,ENVIRON(2))
RESUME(X)== ENVEVAL(X  ,ENVIRON(2))
```

P-74 has the four primitives RUN, RISE, RESUME and ENVEVAL. It also has a system of layers of complexity of process so that activations of modules are layer-1 concepts, activations of recursive programs are layer-2 concepts and, in general, activations of N-layer processes, arranged in a full coroutine regime with extended enveval, are (N + 1) -layer processes. Our four primitives take an extra argument which determines their layer. In this chapter we considered layer-1 concepts. In the next we consider layer-2 concepts. Chapter 9 will describe P-74 in its entirety with an arbitrary number of layers.

P-74 allows the user complete freedom to access and update control structure and to construct activation records. This facility is not generally present in other systems. The Bobrow-Wegbreit system comes nearest to it but places restrictions to allow efficient implementation.

The 1-layer regimes discussed in this chapter may be summarised as follows.

Dendrarchy

Primitives are CALL and RETURN

Relationships are subroutine

Languages using this regime are ALGOL-60,LISP, POP-2

Semi-coroutine

Primitives are RUN,RISE and NEW

Relationships are semi-coroutine

SIMULA 67 includes this regime but only at layer 2

Coroutine

Primitives are RESUME

Relationships are coroutine

This regime is used for many machine code programs at layer 1 and in SIMULA at layer 2

Full coroutine

This is a combination of coroutine and semi-coroutine

Enveval with copy

Primitives are ENVEVAL and ENVIRON

Relationships are enveval

The Bobrow-Wegbreit system uses this regime as does POPLER 1.5

Extended enveval without copy

P-74 includes this regime.

## Chapter 8. 2-processes

All the relationships we have considered so far have been between modules of code. These are basically flowcharts. An activation of such a module, when temporarily halted, only needs to know the next instruction of its flowchart if it is ever to continue execution. There is a more complex situation when two programs, each running under a dendrarchy regime itself, are related in a subroutine, semicoroutine, or coroutine relationship. For example, we may have two recursive ALGOL game-playing programs and we might wish to play them against each other. When either program halts temporarily it must save its entire current run time structure, this being a chain of activation records. Just as modules may require values of variables to be saved on their activation records, so may the two programs. In particular, if each program ran under a regime which allowed it to store activation records in variables, the activation record for the program might implicitly store a more complex control structure than its return chain.

In this chapter we deal with relationships between programs which already have one layer of control structure, and to do this we first compare SIMULA with P-74. We also consider control primitives for jumpout, generalised jumps, and backtracking. P-74 allows these to be set in a new light. We refer to activations of modules as 1-activations and to activations of programs composed of modules as 2-activations. In P-74 we make this distinction very clear and we generalise in the obvious way to N-processes.

SIMULA 67 allows a full coroutine regime between programs, each of which allows a dendrarchy regime between modules. The primitives in SIMULA are CALL, DETACH, RESUME, and NEW. They are related to the P-74 primitives: CALL corresponds to RUN between 2-activations; RESUME corresponds to our RESUME between 2-activations, and DETACH corresponds to RISE between 2-activations. We will in future add a postfix to our primitives making it clear to which layer of activation they relate. Thus we have RUN-1 and so on. Within any one of its one-layer programs, SIMULA allows standard ALGOL-60 block entry and exit.

## 8.1 A comparison of SIMULA 67 and P-74

A difference between SIMULA 67 and P-74 is that SIMULA 67 provides a full coroutine regime between one-layer programs each of which runs under only a dendrarchy, whereas P-74 provides more than a full coroutine regime between 2-programs each of which runs under more than a full coroutine regime. P-74 is freer than SIMULA 67. There are other differences which we now consider.

We have to distinguish two different layers of object. 1-layered objects are defined textually in SIMULA by procedure declarations or block declarations and as in ALGOL are delimited by

```
PROCEDURE <NAME> BEGIN.....END,
```

or by

```
BEGIN.....END.
```

Blocks are entered when control reaches the BEGIN instruction and are exited when it reaches the corresponding END. Procedures have names and can be entered by statements which postfix the procedure name with its arguments in parentheses. To enter, execute, and leave a procedure called FOO which takes one argument we could say

```
FOO(1);
```

As we noted in section 7.1, a new activation of FOO is set up to allow recursion. However, on returning from any module its activation record is destroyed.

In P-74, modules are defined textually as in POP-2 and are delimited by

```
FUNCTION <NAME>;...END;
```

or by

```
LAMBDA;...END;
```

These correspond respectively to modules with names and to modules without. An advantage of POP-2 is that modules can be assigned to variables and then passed around by this means. Since modules in P-74 are organised as a full coroutine regime, we have primitives for

running, rising, and resuming. These primitives take a layer number as argument. When modules are being linked this will be 1. The primitives are RUN, RISE, and RESUME. RUN and RESUME also take an activation as argument to specify where control is to be passed. An example of the use of the primitives is

```
FUNCTION F;          FUNCTION G;          FUNCTION H;
PRINT("F");          PRINT("G");          PRINT("H");
RUN(G,1);            RESUME(H,1);         RISE(1);
PRINT("F");          END                  END;
END;

:RUN(F,1);
FGHF:
```

In P-74, any activation which is left may later be continued. In the case of RUN, the activation is stored as a return link automatically. In all three cases it is stored in the global variable CONTINUEE.

RUN and RESUME may be applied to both modules and module activations, so in the first case we make them produce initial module activations. We can also use the primitive INITIAL to produce an initial activation ourselves. If FOO is defined as

```
FUNCTION FOO;RISE(1) END;
```

It simply rises and upon being continued reaches its end and rises again. We could say either

```
INITIAL(FOO)->A;
RUN(A,1);RUN(A,1);
```

or equivalently

```
RUN(FOO,1);
RUN(CONTINUEE,1);
```

In P-74 the extension to 2-layer activations is obvious. We simply use 2 as our layer number rather than 1. Again, an initial 2-layer activation is produced either by the primitive INITIAL, or by RUN and RESUME when they are given a module.

Corresponding to RUN (X, 2), RISE (2), and RESUME (X, 2), are the SIMULA statements CALL (X), DETACH (), and RESUME (X). CALL and RESUME can only be applied to objects called class instances which correspond to 2-activations. They are defined by textual declarations called class declarations. These are delimited by

CLASS <NAME> BEGIN.....END;

and an instance is generated by

NEW <NAME>.

NEW is slightly different from INITIAL for besides creating a new initial activation it also calls the activation. P-74 has an equivalent primitive NEW.

One of the main differences between the handling of 2-layer activations in P-74 and in SIMULA 67 is that P-74 provides a continuee. This can be useful since the new current state might not otherwise know who had called it. In SIMULA this could be arranged by using the primitive THIS which allows any current activation to reference itself. When any activation passed control it could leave itself in some global variable. We found the situation happened often enough to merit being automatic.

In SIMULA 67, arguments may only be passed into a module or a class instance at its very first entry. The arguments are received by the initial activation and assigned to declared parameters of the activation. Thereafter, communication has to be achieved by means of assignments to variables.

We consider argument passing to be merely an instance of communication in general. In P-74, each time an activation is reactivated it gains access to a queue of messages passed to it from its continuer. It also gains access to a queue for outgoing messages through which it can communicate with any activation it reactivates. Parameters are taken from the input queue and can be declared at any point in the code of a module. The input queue is called INMESS and the output queue is called OUTMESS.

We provide special syntax to make argument and result passing easier to specify and we will use this in all future examples. Each run, rise, resume or enveval is enclosed in double angle brackets. For example

```
<<RUN,FOO,LAYER,ARGUMENT-1,....,ARGUMENT-N>>;
<<RISE,LAYER,ARGUMENT-1,...ARGUMENT-N>>;
<<RESUME,FOO,LAYER,ARGUMENT-1,...,ARGUMENT-N>>;
<<ENVEVAL,FOO,ENVIRONMENT,LAYER,ARGUMENT-1,...,ARGUMENT-N>>;
```

The expression between these brackets is evaluated to give a sequence of items as result. The first of these must be RUN, RISE, RESUME or ENVEVAL. All items after the layer correspond to arguments. They are loaded onto the output message queue immediately before control is transferred. Whenever control returns, the input queue will be unloaded and the messages will be returned as the value of the expression. This syntax allows us to nest runs (etc) quite easily. For example, to find the head of the tail of a list and assign it to second-element, we could execute

```
:<<RUN,HEAD,1,<<RUN,TAIL,1,LIST>> >>->SECOND-ELEMENT;
```

As in ALGOL-60, variables in SIMULA 67 are textually bound. This means that the global variables available to a class instance are those which were declared textually outside the class declaration. P-74 variables are dynamically bound as in LISP or POP-2. We have introduced a 2-layer system of identifiers which is more appropriate to the two different layers of process.

If a variable is declared at layer 1 by a declaration such as

```
DECLARE 1 A;
```

The variable, identified by &1A, is said to be local to the current 1-activation. Whenever the activation is current, that variable is accessible. The variable associated with any non-local identifier is accessed by following the control chain at layer 1 until the first occurence of a 1-activation which has the identifier declared locally. The statement of accessibility for layer 2 variables is the same as for variables at layer 1 with 2 replacing 1.

We can make any number of declarations in one DECLARE statement. For each variable declared we supply a layer number and a name. If the layer number is omitted it is taken to be 1 by default. Similarly, if we use a word as an identifier without prefixing it by "&" and a number, it is taken to be a layer 1 variable, so

```
DECLARE 1 A 2 B C 2 A;
```

The primitive argument can be used to take items from the input queue of an activation. So

```
ARGUMENT 2 A B;
```

Will take two items from the input queue, assigning the first to &2A and the second to &1B. Similarly, the primitive result will add items to the output queue. So

```
RESULT 2 A B
```

Will put the values of &2A and &1B on the output queue. We use these two primitives in the examples later.

A simple example will illustrate how the scope of variables works at any layer. The example will use layer 1.

```
FUNCTION F;
DECLARE 1 A;1->&1A;
<<RUN,G,1,&1A>>;
<<RUN,PRINT,1,&1A>>;
<<RUN,PRINT,1,&1B>>;
<<RUN,H,1>>;
<<RUN,PRINT,1,&1A>>
END;
```

```
FUNCTION G;
DECLARE 1 B;
ARGUMENT 1 B;
<<RUN,PRINT,1,&1A>>;
&1A + &1B -> &1A;
<<RISE,1>>
END;
```

```
FUNCTION H;
DECLARE 1 A;
3-> &1A;
<<RISE,1>>
END;
```

```
:<<RUN,F,1>>;
 1 2 UNDEF 2:
```

F declares a local variable A and assigns 1 to it. It then runs G with the value of A as argument. G declares a local variable B. G prints A and although A is not declared locally, F is on G's return chain so F's version of A is accessed. G side-effects this so that when control

rises to F, A prints out as 2. F prints out B as UNDEF since no declaration of it exists in its control chain. F then runs H. Even though H assigns to A, since A is local to H, F's version is unaltered.

The next example shows how variables of different layers interact.

```
FUNCTION F;
DECLARE 2 THRESHOLD 2 CONSTANT 1 N;
ARGUMENT 2 THRESHOLD 2 CONSTANT 1 N;
<<RUN,G,2,N>>;
LOOP:
2 * &2THRESHOLD -> &2THRESHOLD;
N + &2CONSTANT -> &2CONSTANT;
<<RUN,CONTINUEE,2>>
END;

FUNCTION G;
DECLARE 1 N;ARGUMENT 1 N;
IF N > &2THRESHOLD
THEN <<RISE,2>>
CLOSE;
<<RUN,PRINT,1,N>>;
<<RUN,G,1,N + &2CONSTANT>>
END;

:<<RUN,F,2,10,2,1>>;
 1 3 5 7 9 12 15 18 22 INTERRUPT:
```

F runs G which prints N repeatedly, adding CONSTANT to it each time until THRESHOLD is reached. G is recursive. Since THRESHOLD and CONSTANT are declared at layer 2, they are accessible from G even though G was initially run at layer 2. Any activation of G has a 1-return pointer to the previous activation of G, and a 2-return pointer to F. At THRESHOLD, G rises out of all calls of itself to F by means of a rise-2. F doubles the threshold and increases the constant by N. This N has not been side-effected by G. Since G was run from F at layer 2, N became inaccessible and its value was passed through as an argument.

## 8.2 Examples of the full coroutine regime in P-74

In the literature, full coroutine regimes are generally used in three types of application.

1) Character stream processing (Conway,1963)
2) Dynamic data structures (Dahl,1972),(Hoare,1972)
3) Simulation (Dahl & Nygaard,1966)

We will give four examples written in P-74 and using its first two layers. The first three examples are concerned with character stream processing and the final one with dynamic data-structures. We give no examples of simulation as these would require a discussion of primitives for sequencing processes according to a simulated time scale.

### 8.2.1 Building bricks example

The characters in our first stream-processing example are building bricks rather than alpha-numeric characters. Suppose we have a coroutine called SUPPLY-BRICK which has access to a stock-pile and which returns a brick at random from this stock-pile whenever it is run. In our example, the bricks are of two types, large and small. We have a coroutine called SORTER which accepts orders of the form "LARGE" or "SMALL" and returns a brick of the required type. SORTER keeps two stock-piles itself and when one of them is empty it calls SORT-BRICK until it is given a brick of the right size. SORT-BRICK calls SUPPLY-BRICK and puts the brick that is returned into the correct pile. The code for SORTER is

```
   FUNCTION SORTER;
1) DECLARE LARGEQ SMALLQ ORDER CALLER
2) EMPTYQ->LARGEQ;EMPTYQ->SMALLQ;
3) LOOP: ARGUMENT ORDER;CONTINUEE->CALLER;
   IF ORDER="LARGE"
4) THEN LOOPIF <<RUN,EMPTY,1,LARGEQ>>
        THEN <<RUN,SORT-BRICK,1>>
        CLOSE;
5)      <<RESUME,CALLER,2,<<RUN,POP,1,LARGEQ>> >>
6) ELSE LOOPIF <<RUN,EMPTY,1,SMALLQ>>
        THEN <<RUN,SORT-BRICK,1>>
        CLOSE;
7)      <<RESUME,CALLER,2,<<RUN,POP,1,SMALLQ>> >>
   CLOSE;
8) GOTO LOOP
   END;
```

Comments:

LOOPIF means the same as ALGOL's WHILE loop.

1) SORTER has four layer-1 local variables declared here.
2) Empty queues of bricks are assigned to the stock-piles of large and small bricks.
3) Every time anyone passes control to SORTER, this line will be executed. An argument passed in by the caller will be stored in ORDER and the continuation of the caller will be stored in CALLER.
4) If the order was for a large brick then SORTER will run SORT-BRICK until there is a brick on the large brick stock-pile. SORT-BRICK adds a brick to the relevant stock-pile, and EMPTYQ tests to see if a particular stock-pile is empty.
5) POP is run to take the top brick from LARGEQ. this brick is passed to the caller as an argument. CALLER is run at layer 2. We will explain this later.
6) & 7) are similar to 4) & 5) but for small bricks.
8) When control next returns it will be directed immediately to LP.

Suppose three bricks are required for building an arch, one large and two small. If we have a function BUILD-ARCH which takes three appropriate bricks as arguments and builds an arch from them, we can write a coroutine MAKE-ARCHES which will gather materials and build an endless number of arches.

```
   FUNCTION MAKE-ARCHES;
   DECLARE SUPPORT1 SUPPORT2 BEAM SORTER;
1) ARGUMENT SORTER;
2) LOOP:<<RESUME,SORTER,2,"SMALL">>->SUPPORT1;
   <<RESUME,SORTER,2,"SMALL">>->SUPPORT2;
   <<RESUME,SORTER,2,"LARGE">>->BEAM;
3) <<RUN,BUILD-ARCH,1,SUPPORT1,SUPPORT2,BEAM>>;
   GOTO LOOP
   END;
```

Comments:

1) When MAKE-ARCHES is first run it is given an initial activation of SORTER from which it obtains its bricks.
2) MAKE-ARCHES then goes into an infinite loop. Each time round this loop it asks SORTER for three bricks.
3) It then builds an arch with these bricks.

We can begin executing the program by typing

```
<<RUN,MAKE-ARCHES,2,INITIAL(SORTER)>>;
```

MAKE-ARCHES resumes SORTER at layer two. In this particular example this was unnecessary. However, it becomes so if we alter the problem in various ways. Firstly, we might have a more complex sorter so that each time it resumes MAKE-ARCHES it has to remember its control chain. We could make SORTER search various possible brickyards in a recursive depth-first manner, searching all towns, and for each of these searching all streets, and for each of these trying all brickyards. Alternatively, MAKE-ARCH could be a more complex program. Suppose we used MAKE-HOUSE which entailed making four walls each of which entailed making twenty layers of bricks. Whenever MAKE-HOUSE needed a brick it would need to remember its current control structure.

### 8.2.2 Fringe example

This example of character stream processing requires two layers. We write a semi-coroutine which traverses the fringe of a binary tree from left to right. Each time it is run at layer two it returns the next fringe element. More generally, we shall first write a function which applies an arbitrary function to every fringe element of a

binary tree. The arbitrary function F and the tree are given as arguments to it. By suitable choice of F we can make an instance of the more general function which returns the fringe elements one at a time. We will use the operator :: to combine a left subtree and a right subtree into a binary tree. LEFT is a function which returns the left subtree, RIGHT returns the right subtree, and ATOM tests to see if a tree is simply a fringe element. The code for fringe is

```
   FUNCTION FRINGE;
1) DECLARE F TREE;ARGUMENT F TREE;
   IF <<RUN,ATOM,1,TREE>>
2) THEN <<RUN,F,1,TREE>>
3) ELSE <<RUN,FRINGE,1,F,<<RUN,LEFT,1,TREE>> >>;ERASE();
        <<RUN,FRINGE,1,F,<<RUN,RIGHT,1,TREE>> >>;ERASE()
   CLOSE;
4) <<RISE,1,"DONE">>
   END;
```

Comments:

1) FRINGE has two arguments, F and TREE.
2) The tree is an atom and hence is a fringe element. We apply F to it.
3) The tree is not an atom so we run fringe with F and the left sub-tree as arguments, and again with f and the right sub-tree as arguments. We must erase the message returned from these calls.
4) Control rises out of FRINGE, passing back the message "DONE".

We can use the program to print out the fringe elements as follows

```
:<<RUN,FRINGE,1,PRINT,(1::2)::3>>;
1 2 3:
```

If we run FRINGE at layer 2 we can use F to jump-out of the entire FRINGE program producing just one fringe element. By continuing the FRINGE program we can successively generate other elements.

```
FUNCTION ONE-ELEMENT;
DECLARE ELEMENT;ARGUMENT ELEMENT;
<<RISE,2,ELEMENT>>
END;
```

```
:INITIAL(FRINGE)->GENERATOR;
:<<RUN,PRINT,1,<<RUN,GENERATOR,2,ONE-ELEMENT,(1::2)::3>> >>
1
:<<RUN,PRINT,1,<<RUN,GENERATOR,2>> >>;
 2
:<<RUN,PRINT,1,<<RUN,GENERATOR,2>> >>;
 3
:<<RUN,PRINT,1,<<RUN,GENERATOR,2>> >>;
 DONE:
```

We can now see whether two trees have equal fringes. We must write a function that alternately runs generators for each tree and tests the fringes element by element.

```
   FUNCTION EQUAL-FRINGE;
   DECLARE TREE1 TREE2;ARGUMENT TREE1 TREE2;
   DECLARE GEN1 GEN2 ELEMENT1 ELEMENT2;
   INITIAL(FRINGE)->GEN1;INITIAL(FRINGE)->GEN2;
1) <<RUN,GEN1,2,ONE-ELEMENT,TREE1))->ELEMENT;
2) <<RUN,GEN2,2,ONE-ELEMENT,TREE2>>->ELEMENT2;
3) LOOP:IF ELEMENT1="DONE" OR ELEMENT2="DONE"
   THEN IF ELEMENT1=ELEMENT2
        THEN <<RISE,1,TRUE>>
        ELSE <<RISE,1,FALSE>>
        CLOSE
4) ELSEIF ELEMENT1=ELEMENT2
   THEN <<RUN,GEN1,2>>->ELEMENT1;
        <<RUN,GEN2,2>>->ELEMENT2;
        GOTO LOOP
5) ELSE <<RISE,1,FALSE>>
   CLOSE
   END;
```

Comments:

1) & 2) The first time we run GEN1 and GEN2 we must supply them with arguments.
3) If either of the generators return "DONE" then that generator has exhausted its fringe. they must both be exhausted simultaneously for EQUAL-FRINGE to return TRUE.
4) If the fringe elements are equal, both generators are run to produce their next elements and control goes to LOOP.
5) Otherwise EQUAL-FRINGE returns FALSE.

### 8.2.3 Breadth-first search example

So far, the examples could be programmed in SIMULA in almost the same way as in P-74. The main difference would be that P-74 allows argument passing every time an activation is continued whereas SIMULA does not. The next example could not be programmed in SIMULA in the same way as in P-74. This is because we require a semi-coroutine regime at layer one as well as at layer two. The example is similar to the last one except that it applies a function F to every fringe element of a tree in a breadth-first rather than a depth-first manner.

```
   FUNCTION BREADTH-FIRST;
1) DECLARE F TREE GEN1 GEN2;ARGUMENT F TREE;<<RISE,1>>;
   IF <<RUN,ATOM,1,TREE>>
2) THEN <<RUN,F,1,TREE>>;<<RISE,1,"OK">>;<<RISE,1,"DONE">>
3) ELSE INITIAL(BREADTH-FIRST)->GEN1;
        INITIAL(BREADTH-FIRST)->GEN2;
        <<RUN,GEN1,1,<<RUN,LEFT,1,F,TREE>> >>;
        <<RUN,GEN2,1,<<RUN,RIGHT,1,F,TREE>> >>;
4)      <<RISE,1,"OK">>;
5)  LOOP:
        IF GEN1="DONE" AND GEN2="DONE"
        THEN <<RISE,1,"DONE">>
        CLOSE;
        IF GEN1="DONE" THEN
        ELSEIF <<RUN,GEN1,1>>="DONE" THEN "DONE"->GEN1
        CLOSE;
        IF GEN2="DONE" THEN
        ELSEIF <<RUN,GEN2,1>>="DONE" THEN "DONE"->GEN2
        CLOSE;
        <<RISE,1,"OK">>;
        GOTO LOOP
   CLOSE
   END;
```

Comments:

1) The first time BREADTH-FIRST is run it takes in two arguments and rises. When control returns, BREADTH-FIRST will check if TREE is a fringe element.

2) If TREE is a fringe element, F will be applied to it and BREADTH-FIRST will rise with the result "OK". Next time it is run it will immediately return with the result "DONE".
3) If TREE is not a fringe element, BREADTH-FIRST will initialise two processes GEN1 and GEN2 to search the left-hand and right-hand branches respectively.
4) It will then rise with result "OK".
5) From then on, every time BREADTH-FIRST is continued, it will first continue its left sub-branch program and then continue its right sub-branch program. Thus it applies F to fringe elements on progressively deeper cross-sections of the tree. when any sub-branch is "DONE" this is noted. When both are "DONE," BREADTH-FIRST returns "DONE".

We need a function to drive BREADTH-FIRST as it rises back to top-level after treating each row of the tree.

```
   FUNCTION DRIVE-BREADTH-FIRST;
   DECLARE F TREE;ARGUMENT F TREE;
1) <<RUN,BREADTH-FIRST,1,F,TREE>>;
2) LOOP:
   IF <<RUN,CONTINUEE,1>>="DONE"
   THEN <<RISE,1>>
   ELSE GOTO LOOP
   CLOSE
   END;
```

Comments:

1) We initialise BREADTH-FIRST by running it with arguments.
2) CONTINUEE is the continuation of BREADTH-FIRST.

We can execute the program as follows

```
   :<<RUN,DRIVE-BREADTH-FIRST,1,PRINT,(1::2)::3>>;
   3 1 2:
```

As in the previous example we can produce a generator by using ONE-ELEMENT as the function to be applied to each element.

```
:INITIAL(DRIVE-BREADTH-FIRST)->GEN;
:<<RUN,PRINT,1,<<RUN,GEN,2,ONE-ELEMENT,(1::2)::3>> >>;
 3
:<<RUN,PRINT,1,<<RUN,GEN,2>> >>
 1
:
```

It is interesting that if we omit line 4) of BREADTH-FIRST, the program becomes a depth-first search of the fringe. We leave the reader to work this out for himself.

### 8.2.4 Binary counter example

The program in this example provides a binary counter to which we can give two orders. We can ask it to add one to its contents or else we can ask it to print them out. A counter initially consists of a single activation of a module called COLUMN which represents the units column of the counter. It has a variable, NEXT-COLUMN, which points to the next column of the counter. This is set to an initial activation of COLUMN the first time that the units column is run. Whenever the units column needs to carry over, it sends an "ADD-ONE" message to its next column. In this way the counter grows as space is needed. A column carries over 1 to its next column by resuming it. When all the carries are completed, control can rise directly from the most significant column activated. Each resume will have primed the return location of the resumed column to point back to top-level. On the other hand, RUN is used when the message to the units column is "PRINT". This is because the counter must begin printing from the most significant column. Beginning from the least significant, each column runs the next, and when the most significant is reached, each column in turn prints itself and returns back to the next most significant one. The code for COLUMN is

```
   FUNCTION COLUMN;
   DECLARE N ORDER NEXT-COLUMN;
1) INITIAL(COLUMN)->NEXT-COLUMN;0->N;
2) LOOPIF ARGUMENT ORDER,ORDER="PRINT"
   THEN <<RISE,1>>
   CLOSE
3) LOOPIF TRUE
   THEN IF ORDER="ADD-ONE"
        THEN IF N=0 THEN 1->N
             ELSE 0->N;
                  <<RESUME,NEXT-COLUMN,1,"ADD-ONE">>
             CLOSE
        ELSEIF ORDER="PRINT"
        THEN <<RUN,NEXT-COLUMN,1,"PRINT">>;
             <<RUN,PRINT,1,N>>
        CLOSE;
        <<RISE,1>>
   CLOSE
   END;
```

Comments:

1) The first time COLUMN is executed it initialises NEXT-COLUMN and N.
2) This infinite loop catches the print messages sent from previous columns and begins the printing process by rising. It is not used after the first "ADD-ONE" order is received by this column.
3) This infinite loop is used after the first "ADD-ONE" order is received.
4) If the order is to add one, the addition is done and, if necessary, a carry is passed over to the next column.
5) If the order is to print, the next column is told to print first. when it returns, N will be printed.

The program can be used as follows

```
   :INITIAL(COLUMN)->COUNTER
   :<<RUN,COUNTER,1,"ADD-ONE">>;<<RUN,COUNTER,1,"PRINT">>;
    1
   :<<RUN,COUNTER,1,"ADD-ONE">>;<<RUN,COUNTER,1,"PRINT">>;
    10:
```

We would need to alter the program to have two layers if we wanted counters to be capable of returning their digits one at a time beginning with the most significant digit.

## 8.3 A summary of the 2-layer P-74 primitives RUN,RISE, and RESUME.

At any time there is a current 1-activation which corresponds to the module being executed, and a current 2-activation which corresponds to the program being executed. Any 2-activation has a 2-return address which is either a 2-activation or is the constant NIL. It also has a 2-dictionary to store local 2-variables, and has a pointer to its inner 1-activation which is the current 1-activation within the 2-activation. Any 1-activation has a 1-return address, which may be a 1-activation or nil, and a 1-dictionary. It also has a slot which stores the program counter for the corresponding module and a slot which stores the module itself. The program counter slot also provides space for temporary storage. The structure of a 2-activation is shown in figure 42.

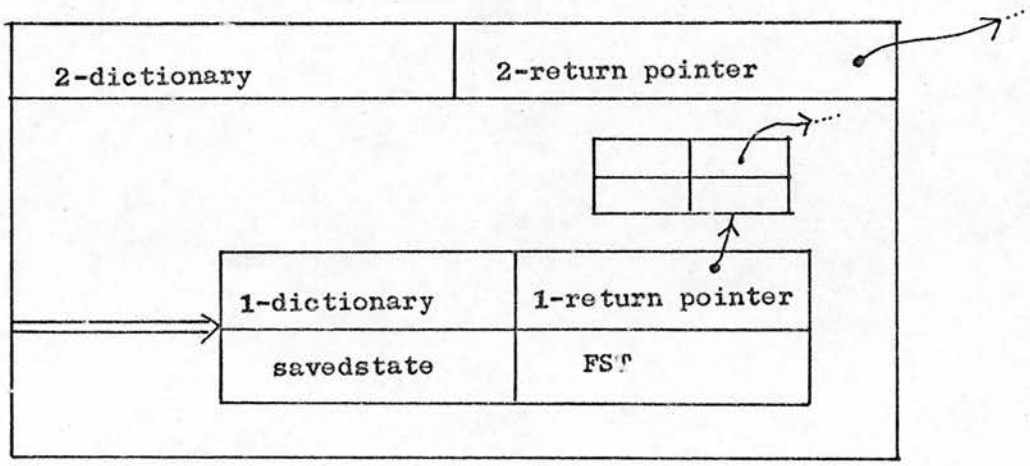



Figure 42. A 2-activation record.

The current 2-activation before execution of a RUN, RISE, or RESUME, is called the tee; the current 2-activation after execution is called the resultant, and the activation to which control is being passed is called the target. The resultant is a combination of target and tee. The continuation of the activation that control has left is called the continuee. At layer 1, target and continuee are 1-activations; at layer 2, they are 2-activations.

Definitions

We shall use the notation INNER (2, X) of a 2-activation to mean the 2-activation itself, and INNER (1, X) of a 2-activation to mean the inner 1-activation of X. INNER (1, X) of a 1-activation X means X itself. For any I, RET (I, X) means the I-return address of INNER (I, X) and DIC (I, X) means the I-dictionary of INNER (I, X).

RUN-1

The target is specified as an argument to the run and is a 1-activation. after a run-1

```
INNER(2,RESULTANT)==INNER(2,TEE)
INNER(1,RESULTANT)==INNER(1,TARGET)
RET  (2,RESULTANT)==RET  2,TEE)
RET  (1,RESULTANT)==INNER(1,TEE)
DIC  (2,RESULTANT)==DIC  (2,TEE)
DIC  (1,RESULTANT)==DIC  (1,TARGET)
CONTINUEE          ==INNER(1,TEE)
```

RUN-2

The target is specified as an argument to the run and is a 2-activation. after a run-2

```
INNER(2,RESULTANT)==INNER(2,TARGET)
INNER(1,RESULTANT)==INNER(1,TARGET)
RET  (2,RESULTANT)==INNER(2,TEE)
RET  (1,RESULTANT)==RET  (1,TARGET)
DIC  (2,RESULTANT)==DIC  (2,TARGET)
DIC  (1,RESULTANT)==DIC  (1,TARGET)
CONTINUEE          ==INNER(2,TEE)
```

RISE-1

---

The target is RET(1,TEE). After a rise-1

```
INNER(2,RESULTANT)==INNER(2,TEE)
INNER(1,RESULTANT)==INNER(1,TARGET)==RET(1,TEE)
RET  (2,RESULTANT)==RET  (2,TEE)
RET  (1,RESULTANT)==RET  (1,TARGET)==RET(1,RET(1,TEE))
DIC  (2,RESULTANT)==DIC  (2,TEE)
DIC  (1,RESULTANT)==DIC  (1,TARGET)==DIC(1,RET(1,TEE))
CONTINUEE          ==INNER(1,TEE)
```

RISE-2

---

The target is RET(2,TEE). after a rise-2

```
INNER(2,RESULTANT)==INNER(2,TARGET)==RET(2,TEE)
INNER(1,RESULTANT)==INNER(1,TARGET)==INNER(1,RET(2,TEE))
RET  (2,RESULTANT)==RET  (2,TARGET)==RET(2,RET(2,TEE))
RET  (1,RESULTANT)==RET  (1,TARGET)==RET(1,RET(2,TEE))
DIC  (2,RESULTANT)==DIC  (2,TARGET)==DIC(2,RET(2,TEE))
DIC  (1,RESULTANT)==DIC  (1,TARGET)==DIC(1,RET(2,TEE))
CONTINUEE          ==INNER(2,TEE)
```

RESUME-1

---

The target is specified as an argument to the resume and is a 1-activation.After a resume-1

```
INNER(2,RESULTANT)==INNER(2,TEE)
INNER(1,RESULTANT)==INNER(1,TARGET)
RET  (2,RESULTANT)==RET  (2,TEE)
RET  (1,RESULTANT)==RET  (1,TEE)
DIC  (2,RESULTANT)==DIC  (2,TEE)
DIC  (1,RESULTANT)==DIC  (1,TARGET)
CONTINUEE          ==INNER(1,TEE)
```

RESUME-2

---

The target is specified as an argument to the resume and is a 2-activation.After a resume-2

```
INNER(2,RESULTANT)==INNER(2,TARGET)
INNER(1,RESULTANT)==INNER(1,TARGET)
RET  (2,RESULTANT)==RET  (2,TEE)
RET  (1,RESULTANT)==RET  (1,TARGET)
DIC  (2,RESULTANT)==DIC  (2,TARGET)
DIC  (1,RESULTANT)==DIC  (1,TARGET)
CONTINUEE          ==INNER(2,TEE)
```

## 8.4 Relationship of layers to jump-out

A first attempt at writing a recursive list processing function to search a tree for particular fringe elements which satisfied a certain predicate, P, might be

```
FUNCTION SEARCH TREE;
IF ATOM(TREE)
THEN IF P(TREE)
  THEN SUCCEED(TREE)
  ELSE
  CLOSE
ELSE SEARCH(HEAD(TREE));SEARCH(TAIL(TREE))
CLOSE
END;
```

The tree is represented as a list structure. Every node is either atomic or else has a HEAD which is the left-hand sub-tree, and a TAIL which is the right-hand sub-tree. The search is done recursively on the sub-branches of the node. When a terminal node, recognised by ATOM, is found, the test predicate P is applied to it. The problem is how the first successful value found is to be returned all the way back to top-level. One way of doing this requires SEARCH to produce a truth value to say whether it succeeded. If the value is TRUE it should also return the successful node. The truth value will be used

to decide whether to carry on the search or to pass the successful result back up to top-level. So we have

```
FUNCTION SEARCH TREE;
IF ATOM(TREE)
THEN IF P(TREE)
  THEN TREE,TRUE
  ELSE FALSE
  CLOSE
ELSE IF SEARCH(HEAD(TREE)) THEN TRUE RETURN
  ELSE SEARCH(TAIL(TREE))
  CLOSE
CLOSE
END;
```

We shall refer to this step by step method of exiting a nest of activations as wind-out. Although wind-out is always possible it is cumbersome and Landin (1965) has suggested a convenient alternative. He suggested that a special function J (for jump-out) should be provided, to which dynamically nested functions could return immediately. We can cause a return by execution of a function, say JUMP. Now our program can be written as

```
FUNCTION SEARCHTOP TREE;
J(LAMBDA;SEARCH(TREE) END)
END

FUNCTION SEARCH TREE
........JUMP();....
END;
```

The jump-out facility has some ideas in common with the coroutine and semi-coroutine relationships between 2-processes. Our search could be written in P-74 for example, as

```
   FUNCTION SEARCH;
1) DECLARE 1 NODE;
2) ARGUMENT 1 NODE;
3) IF <<RUN,ATOM,1,&1NODE>>
4) THEN IF <<RUN,P,1,&1NODE>>
5)      THEN <<RISE,2,&1NODE>>
        CLOSE
6) ELSE <<RUN,SEARCH,1,<<RUN,HEAD,1,&1NODE>> >>;
7)      <<RUN,SEARCH,1,<<RUN,TAIL,1,&1NODE>> >>
   CLOSE
   END;
```

Comments:

1) NODE is a local variable at layer 1.
2) An argument is taken off the input message queue and is assigned to NODE as the current one to be examined.
3) This checks if the current node is terminal.
4) This checks the terminal node using the predicate P.
5) Control rises out of all layer 1 calls of SEARCH back to the top-level call at layer 2.
6) SEARCH is run at layer 1 with the head of the current node as argument (i.e. the left-hand sub-tree).
7) SEARCH is run at layer 1 with the tail of the current node as argument (i.e. the right-hand sub-tree).

To search a tree we execute the top-level statement

```
<<RUN,SEARCH,2,TREE>>;
```

When control returns, if a result is returned it will be the fringe element found. Otherwise, the search will have failed. RISE-2 is used to jump out of all the runs of SEARCH which were made at layer 1.

There is an important difference between jump-out and RISE-2. RISE-2 leaves an activation record in CONTINUEE which can be continued. If we now execute

```
<<RUN,CONTINUEE,2>>
```

Search will continue from where it left off and will return the next fringe element satisfying P. In other words, RUN and RISE at layer 2 provide jump-out and jump-in for 2-layered processes. Although it can be argued that jump-out is only a convenience and not strictly necessary this cannot be argued for jump-in. There is no process corresponding to wind-out for winding-in under a dendrarchy. Wind-in would be possible under a semi-coroutine regime between 1-activations. However, both wind-in and wind-out are cumbersome when compared with RUN-2 and RISE-2.

There is another difference between RISE and jump-out. From a particular depth of recursion it is possible to jump-out to various points higher in the recursion. After making these jumps, since control does not necessarily return to top-level, it may still be possible to jump yet further out. The chain of return links is thought of as one long string, as illustrated in figure 43.

In P-74 at layer 2, the system of return links does not look like a string. Return pointers at layer 2 form a string of activation records each of which refers to a string of layer 1 return pointers. A situation like this is set up if we execute

```
<<RUN,A,3,3>>
```

with the code shown below.

```
FUNCTION A;
DECLARE N;ARGUMENT N;
IF N IS NOT EQUAL TO 0
THEN <<RUN,A,1,N-1>>
ELSE <<RUN,<<RUN,B,2,3>>,2>>
CLOSE
END;

FUNCTION B;
DECLARE N;ARGUMENT N;
IF N IS NOT EQUAL TO 0
THEN <<RUN,B,1,N-1>>
ELSE <<RUN,C,2,3>>;<<RISE,2,CONTINUEE>>
CLOSE
END;

FUNCTION C;
DECLARE N;ARGUMENT N;
IF N IS NOT EQUAL TO 0
THEN <<RUN,C,1,N-1>>
ELSE <<RISE,3>>;
    <<RISE,2>>;
    <<RISE,3>>
CLOSE
END;
```

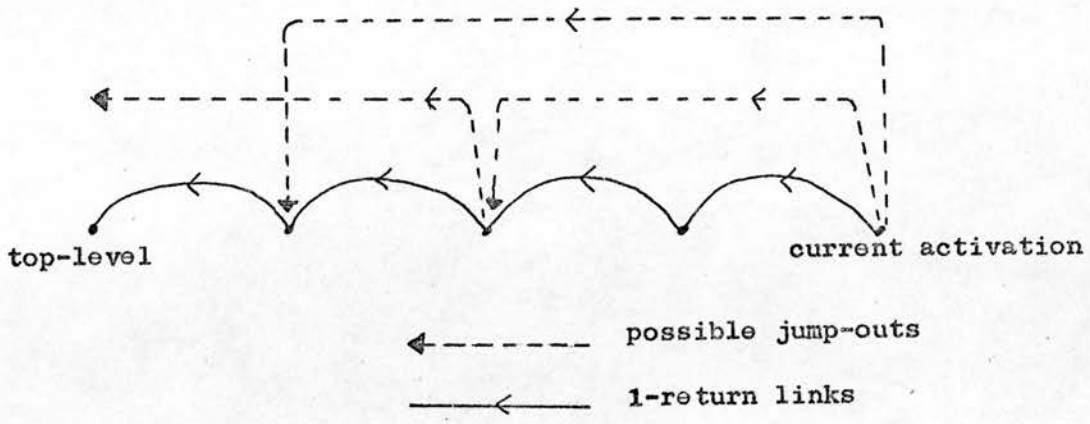



Figure 43.

The RISE-3 is simply used to bring control back to top-level from being nested in several RUN-2's. The control links are shown in figure 44.

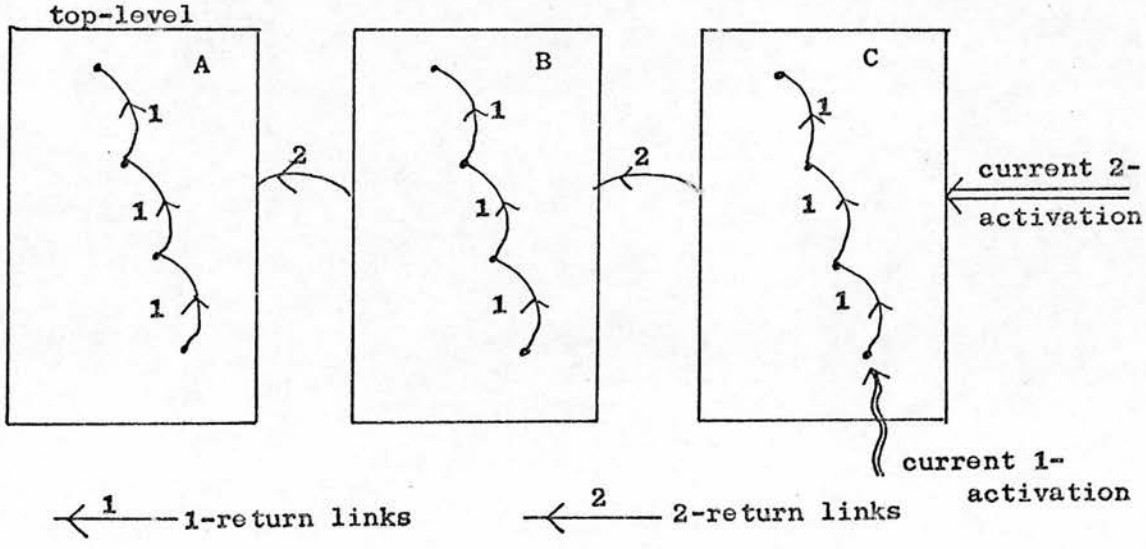



Figure 44.

Here the return pointers form a tree. The one layer activations in B are not to be regarded as above the 1-activations in C. They are part of a separate 2-activation which happens to be above the 2-activation C at the moment. Since A, B, and C of figure 44 are part of a full-coroutine regime, C may RISE-2 to B which may RISE (2) to A, passing back C's continuation. Then A may RUN-2 C's continuation. This is the case if we continue running our example by typing

```
:<<RUN,CONTINUEE,3>>;
```

The situation at the next RISE-3 is shown in figure 45.

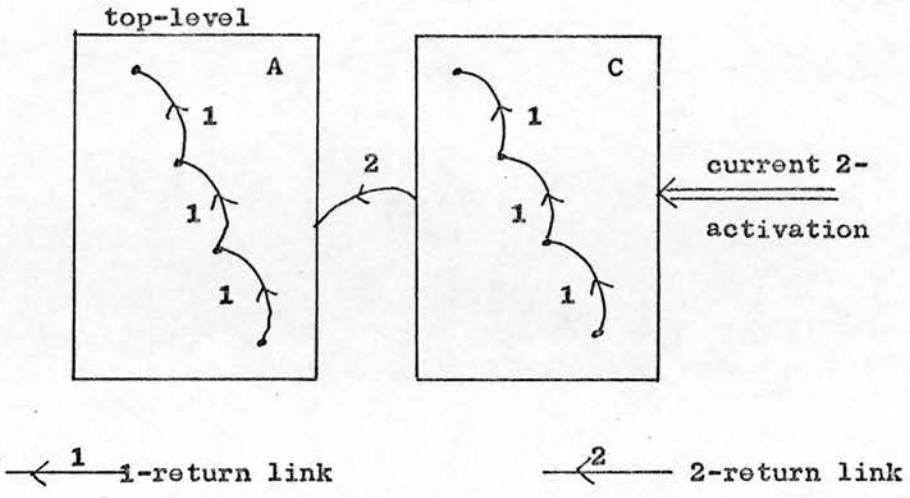



Figure 45.

If we had considered the 1-activations in A, B, and C as a single chain, its central part would be removed. The chain could clearly not be a stack. In semi-coroutine regimes, it is the ability to treat activations as objects that allows this behaviour. Otherwise, the different layers of SIMULA and P-74 dissolve into one.

## 8.5 Generalised jumps

Generalised jumps are a generalised version of the LABEL facility of ALGOL-60. In ALGOL-60 it is possible for control, when nested several modules deep in execution, to be passed to a labelled statement in any module currently on the control return chain. In this respect, jump to labels correspond to the jump-out facility in POP-2 and to Landin's jump-out. On the other hand, labels can be used as objects in ALGOL-60 and can be assigned to variables and thereby passed around.

If we remove the restriction that labels can only be jumped to while their module's activation record is in the current control environment then we extend the facility to generalised jumps. A control regime like that in ALGOL-60, where activations are only available while still on the current control environment, is clearly inadequate for this extended facility. In a semi-coroutine regime, an activation record can outlive the exit of its corresponding module, and generalised jumps are possible with a little modification of the regime. Let us see what modification this is.

Suppose module F runs module G in a 1-layer semi-coroutine regime. Furthermore, G reaches a statement which requires labelling. The current activation is available at this point but cannot directly be used to represent the label since it will subsequently be side-effected as control proceeds beyond the statement labelled in G. The activation really needs copying to retain the program counter's value at the labelled statement. Suppose we use a new primitive called TAG to produce a copy, G', of the current activation. Suppose further that G now rises to F which runs H which runs J, and that J wishes to jump to the label produced by TAG. This is the situation shown in figure 46.

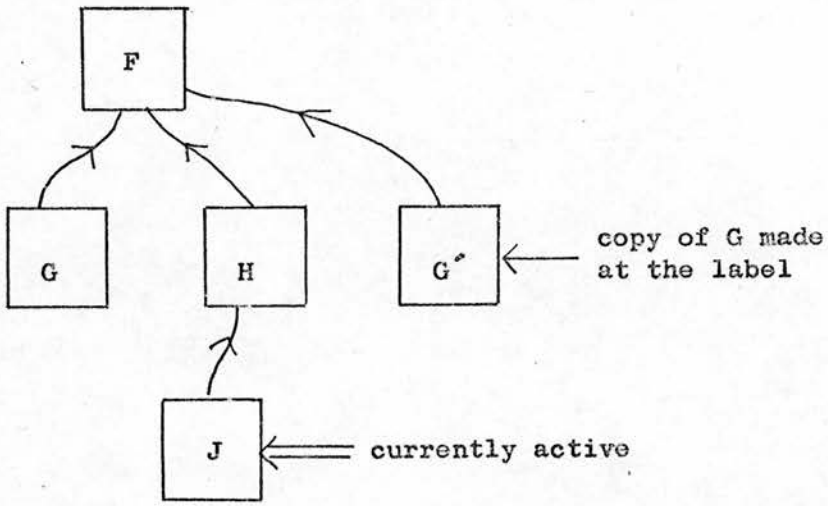



Figure 46.

In a semi-coroutine regime, the only primitives that might achieve this are RUN and RESUME but neither of these will work. When they continue the copied activation of G, they coerce its return activation to be different from F. If J runs G' then the return link will be to H and if J resumes G' then the return link will be to F. We need a new primitive which will continue execution of G while retaining its return link to F. In Conniver this primitive is called GO. GO is a particular case of the ENVEVAL of Bobrow and Wegbreit where the form to be evaluated is vacuous. Thus

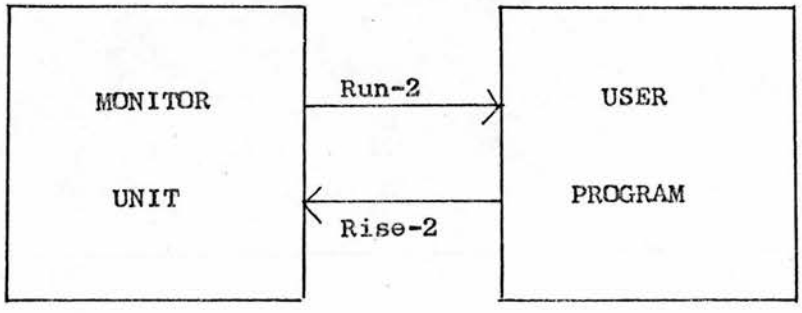
```
GO(X)==ENVEVAL(IDENTFN,X)
```

In this model of generalised jumps, TAG still might not copy the entire control chain. Although the return chain of activation records is the same when control returns to G as when the tag was constructed,

some of the activation records on this chain might have been side-effected. On exiting G, control returned to F and F may have done some computing before passing control to H. In an enveval with copying regime this problem would not arise. In a semi-coroutine regime even the control chain may be different. If the control chain had been A, B, C, D, E, F, G for example, and D had been run from a module S at the bottom of a control chain P, Q, R, S, then D would have had its return activation coerced to be S and the return chain of G would now be P, Q, R, S, D, E, F, G.

Thus, to provide fully for generalised jumps we must provide copying of return chains of activations. The same problems arise when we discuss backtracking.

Generalised jumps can be expressed in a conceptually clearer way in P-74 since we can use two layers to describe them. Consider two 2-layered processes, one being the user program which uses generalised jumps, and the other being a monitor program. Let's assume that the monitor and user program are in a semicoroutine relationship so that the monitor runs the user program at layer 2 and the user program rises at layer 2 back to the monitor program. We could equally well have employed a coroutine relationship between the two. Remember that a run-2 gives the user program's 2-activation a layer-2 return link to a 2-layered activation complete with its current stack of 1-activations. Similarly, a rise-2 to the monitor leaves the current 2-layer snapshot of the user program in the variable CONTINUEE. The situation is illustrated in figure 47.




The user program and the monitor unit are each written using RUN-1 and RISE-1. They communicate using RUN-2 and RISE-2.

Figure 47.

When the user program wishes to save its control state by making a label, it runs the primitive TAG at layer 1. TAG will rise at layer 2 to the monitor program and send it the message "TAG". On receiving this message the monitor program will copy the continuee and run it at layer 2, sending the copy as message. TAG will receive this message and rise-1 with it as result.

When the user program wishes to jump to a tag, it will run the primitive GO at layer 1 with the tag as message. GO will rise at layer 2 to the monitor with a message consisting of the tag. The monitor will simply run the tag at layer 2 and pass back a further copy of the tag as message.

The code for the monitor and TAG and GO are given below. The monitor is initially run at layer 1 with the user program as argument.

```
FUNCTION MONITOR;
DECLARE USER B;
ARGUMENT USER;
<<RUN,USER,2>>->B;
LOOP:
IF B="TAG"
THEN <<RUN,CONTINUEE,2,<<RUN,COPY,1,CONTINUEE>> >>
ELSE <<RUN,<<RUN,COPY,1,B>>,2,B>>
CLOSE;
GOTO LOOP
END;

FUNCTION TAG;
<<RISE,1,<<RISE,2,"TAG">> >>
END;

FUNCTION GO;
DECLARE LABEL;ARGUMENT LABEL;
<<RISE,1,<<RISE,2,LABEL>> >>
END;
```

Although it is a 2-layered system, SIMULA is not able to provide generalised jumps because of the need for copy.

## 8.6 Backtrack;

Floyd (1967), Hewitt (1969), and Golomb and Baumert (1965) have discussed uses of backtracking in problem solving. Bobrow and Wegbreit have shown how a simple use of backtracking can be programmed using their enveval regime. Davies (1973) has extended this enveval regime to allow more complex backtracking with generalised failure actions, and has implemented this in POPLER 1.5.

Backtrack originated from the idea of non-deterministic programming. In a non-deterministic program there are certain choice points at which the program is supposed to fork, each branch of the fork corresponding to a particular choice. The branches are assumed to be entirely independent so it is as though several distinct copies of the state of the program come into existence at the fork. Some of the branches may reach a failure point and die out. Others may terminate successfully with a result. The original program is said to have a set of results each correponding to a successful path through the choice tree. If the choice tree is searched depth first then we have a backtrack regime.

There are two pure approaches to the implementation of a backtrack system. In the approach called Floyd backtracking a record of every change in state of the program is kept on a stack. When any branch of the search tree reaches a failure this stack is unwound with all the actions being done in reverse until a choice point is backed up to. At this stage the state of the program will have been restored. The alternative approach is to save the complete state of the program at choice points and to store these on a stack. At a failure, the topmost state on the stack must be reinstated and the next choice taken.

In practice, not everything need be remembered, and either of these two methods can be used in an impure form. Alternatively, a combination of them may be used. There are three particular categories of information which may be restored at a failure. These are

1) The current control environment. In a flowchart this would simply be the program counter. In a recursive program it would also include the return chain.
2) The access environment. This includes all variables current at the time of the last choice point.
3) Contents of data structures accessible through the access environment.

Information of the third category is almost always reinstated by the Floyd method. In POP-2 the library backtrack program is based on a primitive which saves both control and access environment by the copying method. In POPLER 1.5, the control environment is copied and the access environment is restored by the Floyd method. In the present discussion we confine ourselves to questions of saving and restoring control environments.

As with generalised jumps, we will first restrict our discussion to backtracking in a one layered system. In these systems, the control information which needs saving at a choice point consists of the current program counter and the current chain of activation records. We have already defined a primitive TAG which will save precisely this and we have discussed the problems of copying the return chain associated with this primitive. All that remains is to consider how tag, and the associated primitive GO, may be used to implement backtrack. We explain next how backtracking by the method of copying may be implemented using the two primitives.

We require two primitives, one to provide choice points and the other for use when any branch fails. Let us assume we have a global variable whose value is a stack of the control states saved at choice points so far, and let the topmost state on this stack always correspond to the latest choice point visited. Our two primitives which we call CHOICE and FAIL can now be expressed as follows

```
FUNCTION CHOICE;
<<RUN,PUSH,1,<<RUN,TAG,1>>,STACK>>
END;
```

```
FUNCTION FAIL;
<<RUN,GO,1,<<RUN,POP,1,STACK>> >>
END;
```

Where PUSH will add its argument to the top of the stack of control states and POP will remove the top of that stack and return the top as result. CHOICE runs TAG to save the current state and pushes it onto the stack. FAIL runs POP to return the latest saved state and then reinstates this by running GO.

Just as we illustrated generalised jumps by using two layers of process and by having a monitor, so we can do the same with backtracking. The result will give some insight into the nature of backtracking. Instead of considering a backtracking program as a program with two extra primitives which operate on a list of saved control states, it is useful to think of it as a program which communicates with a monitor at two critical times, when a choice point is reached, and when a failure is reached. The advantage of this is that the monitor can clearly be seen as the driving force which controls the search through the non-deterministic choice tree. It is much easier to see how non-depth-first searches of the choice tree might be controlled.

We again have the situation of figure 47 where a monitor program of two layers runs a user program of two layers by runs of layer 2. The user program returns control to the monitor by rising at layer 2 and passes back the message "CHOICE" or "FAIL".

In this case the monitor program will keep track of the saved control states corresponding to the user program's choice points. It will store these on a push-down stack. Whenever the user program tells the monitor that a choice point has been reached, the monitor will have available, in CONTINUEE, the 2-layer activation record for the user program. It will store a copy of this on its stack and return control to the user program telling it, by means of a message, to take the first choice. If the user program informs the monitor that it has reached a failure point, the monitor will reinstate the state corresponding to the last choice point and will pass in a message informing the user to take the second choice this time. We can arrange this with the code

```
   FUNCTION MONITOR;
   DECLARE STACK USER C MESSAGE;
1) EMPTY-STACK ->STACK;
2) ARGUMENT USER; <<RUN,USER,2>>;
3) LOOP: ->MESSAGE;
   IF MESSAGE="FAIL"
4) THEN IF <<RUN,EMPTY,1,STACK>>
        THEN <<RISE,1,"FAIL">>
        ELSE <<RUN,<<RUN,POP,1,STACK>>,2,"SECOND-TIME">>
        CLOSE
   ELSEIF MESSAGE="CHOICE"
5) THEN CONTINUEE->C;
        <<RUN,PUSH,1,<<RUN,COPY,1,C>>,STACK>>;
        <<RUN,C,2,"FIRST-TIME">>
6) ELSE <<RISE,1,"SUCCESS">>
   CLOSE;
   GOTO LOOP
   END;
```

Comments:

1) The stack of saved states is initialised to an empty stack.
2) The user program is taken as an argument and is run-2.
3) Every time the user program returns, its result is put in MESSAGE.
4) The message is "FAIL". If the stack is empty, the monitor returns to top-level with the result "FAIL". Otherwise, the stack is popped and its top saved-state is run with argument "SECOND-TIME" to show that this state has been run before.
5) The message is "CHOICE". A copy of the user program's state (CONTINUEE) is pushed onto the stack and the user program is continued with argument "FIRST-TIME".
6) The user program has succeeded and MONITOR returns to top-level.

In this version the monitor is an iterative program which stores its stack of saved states in a single variable called STATE. Now we have separated out the backtrack monitor from the user program we can, if we wish, make the monitor itself into a recursive program as follows

```
   FUNCTION MONITOR;
   DECLARE STATE MESSAGE;
1) ARGUMENT STATE;
2) <<RUN,<<RUN,COPY,1,STATE>>,2,"FIRST-TIME">>;
3) LOOP: ->MESSAGE;
   IF MESSAGE="CHOICE"
4) THEN <<RUN,MONITOR,1,CONTINUEE>>
   ELSEIF MESSAGE="FAIL"
   THEN IF STATE="EMPTY"
5)      THEN <<RISE,1,"FAIL">>
6)      ELSE <<RUN,STATE,2,"SECOND-TIME">>;
             "EMPTY"->STATE
        CLOSE
   ELSEIF MESSAGE="SUCCEED"
7) THEN <<RISE,2,"SUCCEED">>
   CLOSE;
   GOTO LOOP
   END
```

Comments:

1) MONITOR takes an argument which is the state at a choice point.
2) A copy of the state is run-2.
3) Control can reach here after a return either from the user program or from a lower call of MONITOR. the result of the returner is stored in MESSAGE.
4) If the message was "CHOICE", it came from the user program whose continuee will be the choice point. MONITOR is run recursively to deal with this choice point.
5) The message is "FAIL". If STATE is empty, both choices have been taken already. MONITOR rises to the next higher call of MONITOR with the message "FAIL".
6) The message is "FAIL" and STATE is not empty. STATE is run a second time and is emptied.
7) If the user program succeeds, control jumps-out of all calls of MONITOR with a rise-2 and the result "SUCCEED".

We can execute a user program under this monitor by running the monitor at layer 2 with the user program as its argument. MONITOR will eventually rise-2 with the message "SUCCEED" or "FAIL". If it succeeds, we can continue it to get the next possible path through the non-deterministic user program. We do this by running the continuation at layer 2. Control will jump-in to the nest of MONITOR modules and will carry on from there. Thus

```
:INITAL(MONITOR)->PROGRAM;
:<<RUN,PRINT,1,<<RUN,PROGRAM,2,USER-PROGRAM>> >>;
:SUCCEED
:<<RUN,PRINT,1,<<RUN,PROGRAM,2,"FAIL">> >>;
:FAIL
```

The stack of 1-activations of MONITOR corresponds to the run-time stack of instances of the variable STATE. If the user program has passed through N choice points then the monitor program is nested N deep in recursion. This beautifully illustrates why a backtrack program requires two stacks since both stacks appear on equal status in the program. One is the monitor program's stack and the other is the user program's stack.

## 8.7 The use of ENVEVAL to implement coroutines

We have shown how the Bobrow-Wegbreit ENVEVAL can be modified to make it possible to simulate coroutine and semicoroutine relationships between modules. Bobrow and Wegbreit show how their original ENVEVAL can easily simulate coroutines between 2-activations. They represent a coroutine as a list containing an environment descriptor for the coroutine. Thus the control structure of a coroutine is the chain of activation records from the environment descriptor to top-level. If we have several such coroutines, then since all their control chains lead up to top-level they can easily have common parts as in figure 48.

In a with-copy regime all the paths from a set of environment descriptors would be distinct since common parts would have been copied.

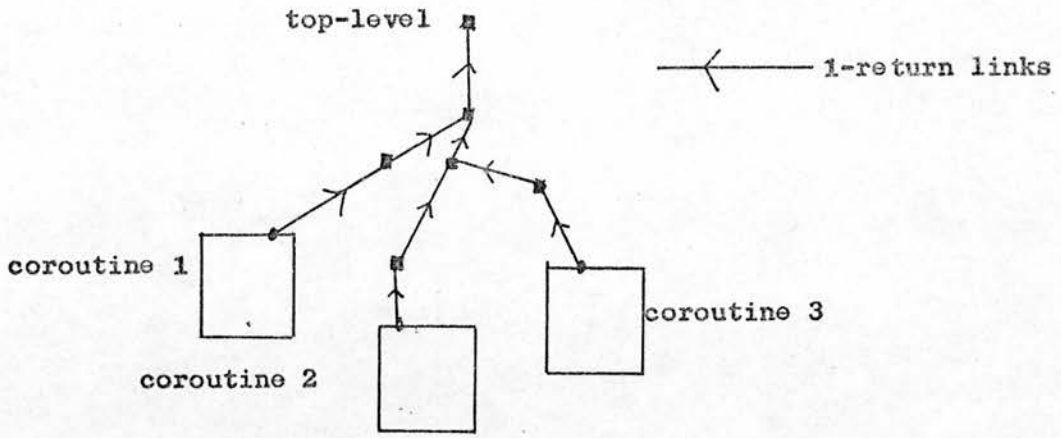



Figure 48. Coroutines sharing control structure.

The primitives START and RESUME can easily be programmed. START simply assigns its argument coroutine to a global variable CURRENT-PROC, and evaluates any given form (perhaps NIL) in the environment stored in the coroutine. RESUME is then able to access CURRENT-PROC to update the current environment descriptor, and having done this can start its argument coroutine.

It is slightly more difficult to implement a semicoroutine regime. Each coroutine now needs to be a list of two elements: an environment descriptor corresponding to the control structure of the coroutine, and a return coroutine. Primitives RUN and RISE can then be programmed to manipulate return coroutines correctly. With this representation resume can also be programmed to allow a full coroutine regime.

Although the enveval regime gives freedom to construct a full coroutine regime as in SIMULA 67, all the work of manipulating return addresses must be done by the programmer. The enveval regime, with its stack implementation, could be thought of as an excellent system for allowing many different more structured regimes to be implemented.

In chapter 7, we saw how an extended ENVEVAL would immediately allow run, rise, and resume relationships between modules. In P-74 we generalise ENVEVAL to have many different layers as we do for other primitives. At layer two, as in fact at all layers, we again have an immediate representation for RUN, RISE, and RESUME. ENVEVAL operates on 2-activations just as at layer one it operated on 1-activations.

We now have the equations

```
RUN(X,2)    ==ENVEVAL(X   ,ENVIRON(1,2),2)
RISE(2)     ==ENVEVAL(NIL,ENVIRON(2,2),2)
RESUME(X,2)==ENVEVAL(X   ,ENVIRON(2,2),2)
```

Where the last arguments of RUN, RISE, RESUME, ENVEVAL and ENVIRON are their layer numbers, in this case 2. In P-74 we use RET instead of ENVIRON. RET applies to any activation, not only the current one.

```
ENVIRON(I,J)==RET(J,RET(J,RET(J,.....RET(J,CURRENT))...))
```

# Chapter 9. A generalisation to arbitrary layer numbers.

In chapters 7 and 8 we discussed several control structures and showed how our concept of layers of control structure related and unified them. Each regime we considered is available in P-74 both at layer one (between modules) and at layer two (between layer-1 programs). This means that the control primitives of P-74 are at least as rich as those in other systems. Since they are unified they are more than just a collection of primitives.

We gave examples written in P-74 to show that two layers can be used fairly easily and intuitively. We also showed that several control ideas, commonly used elsewhere, implicitly use two layers. This is not generally recognised and the distinction between layers is never stated clearly. We explicitly state the concept of layers by making each control primitive take an additional argument specifying its layer number in any particular instance of its use. This has the immediate advantage that layer-1 and layer-2 become very similar to each other. The only difference between them is that layer-2 primitives take precedence over layer-1 primitives. For example, a rise-2 corresponds to the last run-2 executed, regardless of any run-1's or rise-1's that have been executed in the meantime. A rise-1 similarly corresponds to the last run-1, but if a run-2 was executed meantime, there is no corresponding run-1, and a rise-2 would be executed instead. Another way to phrase this is in terms of 1-activations and 2-activations. For example, the current 2-activation remains current no matter how many run-1's or rise-1's are executed, whereas a 1-activation ceases to be current when any layer-1 or layer-2 primitive is executed.

A very obvious and tempting generalisation can now be made to our scheme. It is to allow an arbitrary number of layers. We can allow the use of any positive integer as a layer number for the control primitives, and must arrange that primitives used at layer M have precedence over primitives used at layers less than M. Section 9.1 explains informally the behaviour of P-74's primitives at arbitrary layer numbers, and section 9.2 gives a more formal definition of them.

We expected this generalisation to provide greater expressiveness. This did not happen, for the layers greater than two only seem to

apply to artificially concocted situations. We have been unable to find a convincing example that requires more than two layers. However, this result has a positive aspect. It raises the question why such an obvious generalisation should have little applicability. Our conclusion is that even the use of two layers must be wrong and that a system following the lines of ACTORS (Hewitt, 1973) is needed. In turn, this forces us to explain why languages which use two layers were written. SIMULA is a notable example. We must see how two layer programs can be written in an alternative, layer-free, manner.

We deal with all these questions in section 9.3 which concludes this part of the thesis. However, before we criticise our generalisation any further, let us describe it in detail.

## 9.1 The general P-74 primitives described informally.

Every activation record represents a snapshot of a process and has a number of layers. Layers can be peeled off to reveal subordinate layers called inner layers, or layers can be enveloped around an activation to make an activation with more layers. Every process has an innermost layer and layers are numbered from this innermost layer which is numbered one. This means that all processes existing in the sytem have a common base layer. Hence, there exists a stratification throughout the system.

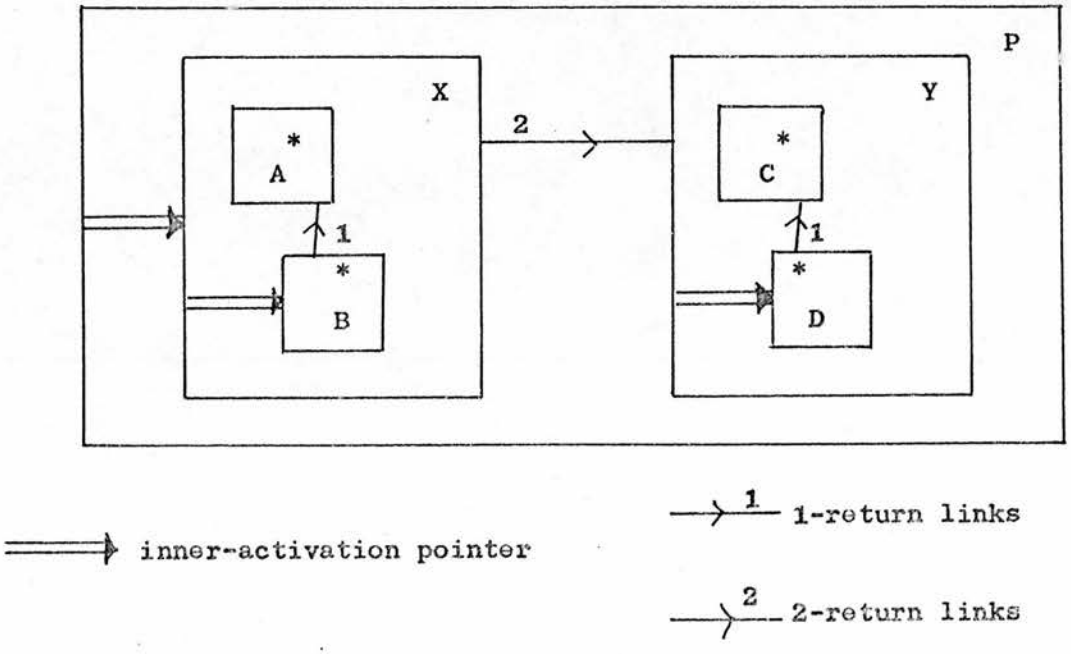



Figure 49.

Figure 49 illustrates a 3-activation, P. Y & X are 2-layered activations. A, B, C, & D are 1-layered activations.

Code (POP-2 text, procedures etc) is only associated with 1-layered activations. In the diagram, the boxes associated with 1-layered activations also represent the text of these activations. The asterisks represent the program counter which refers to the point execution has reached. Every 1-layered activation has such a program counter. No other activations have them.

Since we have no real parallelism, only one 1-layered activation has a currently active program counter (i.e. has its code being executed and its program counter incremented). In our diagram, this is B. B has a return link to A. This link is between two 1-activations and so is a 1-link. B is the currently active 1-activation and X is a currently active 2-process. Return links order activations into a hierarchy. We use the words "above" and "below" to specify direction in this hierarchy, so B is below A.

Layers provide another hierarchy and we will use "inner" and "outer" to denote direction in it. Thus, B is inside X, and X is inside P. There is a 2-link connecting X to Y, so Y is above X. In diagrams we use -> arrows to denote the up/down hierarchy, and => arrows to denote the in/out hierarchy. We can use these arrows to see why B is the current 1-activation in figure 49. We simply follow the => arrows from P inwards. This gives us the current 3, 2 and 1 activation, respectively.

Notice that, since only 1-activations have program counters, the current instruction being executed in B must be regarded as the current instruction being executed in the 2-activation, X. In other words X and B temporarily share program counters. X previously shared program counters with A while A was active.

The hierarchies => and -> are related in an interesting way. In a sense we have more than one -> hierarchy. We have several at each layer. Thus D is lower than C and B is lower than A, but D and B are unrelated. Similarly X is lower than Y but the link is at a different layer from that between D and C, and again X and D are unrelated.

As the P-74 system is currently arranged, only one activation knows

its outer activation although every activation has a pointer to its next inner one. The exception is the current 1-activation, because the current nest of activations is always available in a variable called CUSTATE. We can thus work out the outer activations by chaining inwards from the outer layer of the current nest.

The two types of link -> and => can be created dynamically by the primitives RUN, RISE and RESUME which are generalisations of coroutine CALL, DETACH, and RESUME. In the remainder of this section we informally describe the action of these primitives.

RUN

---

When a currently active activation, P1, runs an activation, P2, it runs it at a particular layer. A resultant activation, composed of P1 and P2, becomes active. If the layer of the run is M, the resultant is formed as follows. The inner M layers (=< M) of the resultant are the inner M layers of P2 and the outer layers (> M) of the resultant are the outer layers of P1. This describes the change in => linkages. The only change in -> linkages is that a new layer-M return link is made, pointing from the resultant to the inner M layers of P1. These layers are now detached from the outer M layers.

Figure 50 shows the situation before and after Y executed a run of X at layer 2. There is one change of => linkages and one addition of -> linkages. The 2-activation Y is now called the continuation of Y, and becomes the layer-2 return activation of X.

New 1-activations can be made from procedures. If such an activation were run at a layer M higher than 1, it would first be enveloped in M-1 layers to make it an M-activation. M-1 new => pointers would be formed.

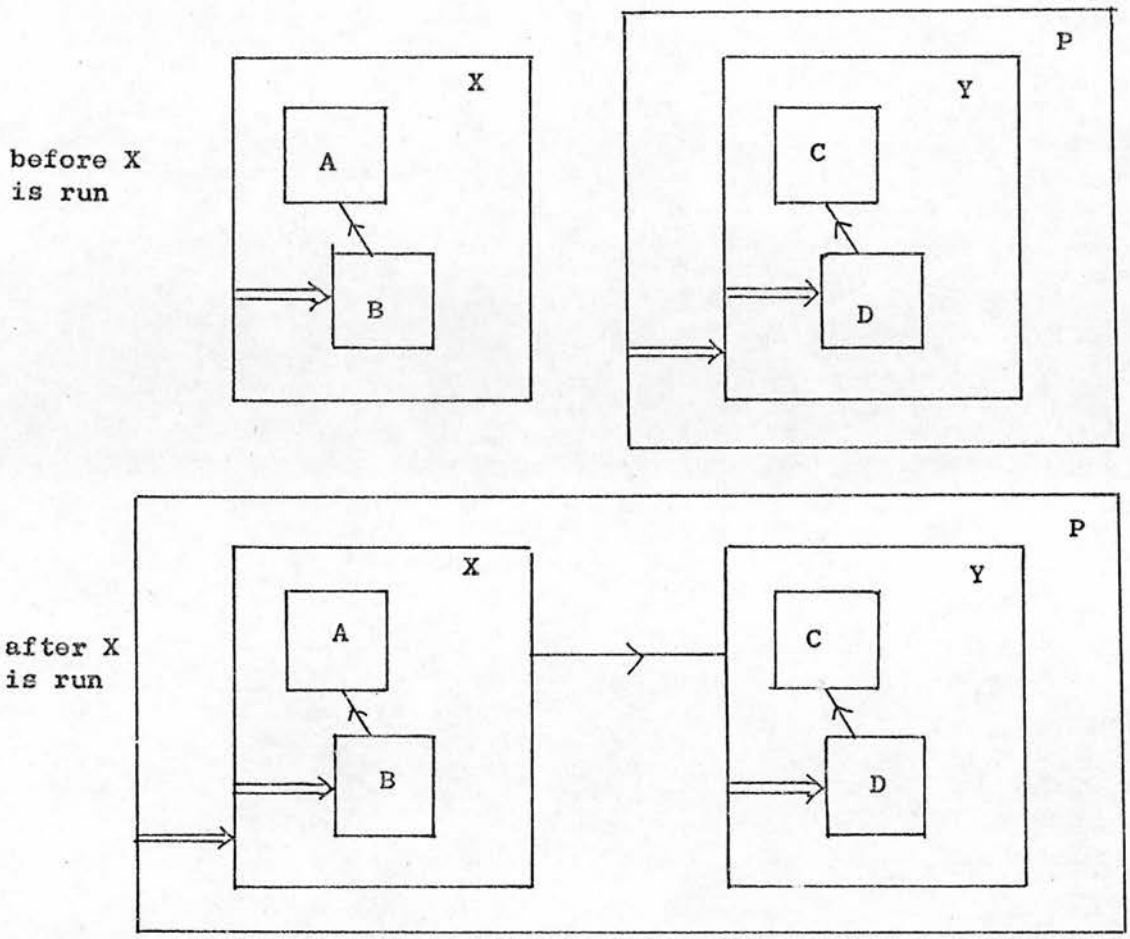



Figure 50.

RISE

---

RUN is an extension of CALL and it sets up a return link from the called activation to its caller. Similarly, RISE is an extension of RETURN and returns to its caller by using the return link that was set up automatically when that activation was last run. If a current activation P1 executes a RISE at layer M, first of all the activation consisting of the inner M layers of the current activation P1 is found. This M-activation will have a -> link to the M-activation P2 which ran it at layer M. Since P2 is a return M-activation it will have exactly M layers. The process of rising from P2 is exactly the same as the process of running P2 from P1 at layer M except that no new return link is made. The continuation of P1 (the inner M layers of P1 that control has left) is made available after the rise in a special variable called CONTINUEE, and will still have its old return link stored in it. If P1 rises at layer M to a return activation P2, P2 may run the continuation of P1. P2 is like a master to P1. He gives P1 control but makes sure that he will return automatically. When

the servant returns, he is made available to the master to be continued as required. Because of the provision for continuation, the relationship between P2 and P1 is a semi-coroutine relationship at layer M, rather than a subroutine relationship.

RESUME

---

At each layer, activations can also be in a coroutine relationship to each other. Suppose an M-activation P1 has M-run an M-activation P2. So P1 is P2's M-master. Then P2 can M-resume an M-activation P3. This now makes P1 the master of P3. If P3 M-rises, he will return control to P1 (see figure 37, section 7. 3). P2 is no longer active and no longer subordinate to P1. He is left available to P3 in the special variable, CONTINUEE. P3 could instead M-resume P2 whereupon P2 would again have P1 as his M-master.

When any M-activation, P, is run or resumed at layer N (where N is not equal to M), P is adjusted to have exactly N layers. If P has too few, N-M layers are enveloped around it. If it has too many, only the inner N are taken.

Side-effects

---

It is design philosophy that an M-activation should be updated whenever it becomes active and that activations should be available to the user as items to be stored or manipulated. In figure 51, two variables A and B refer to the same 2-activation.

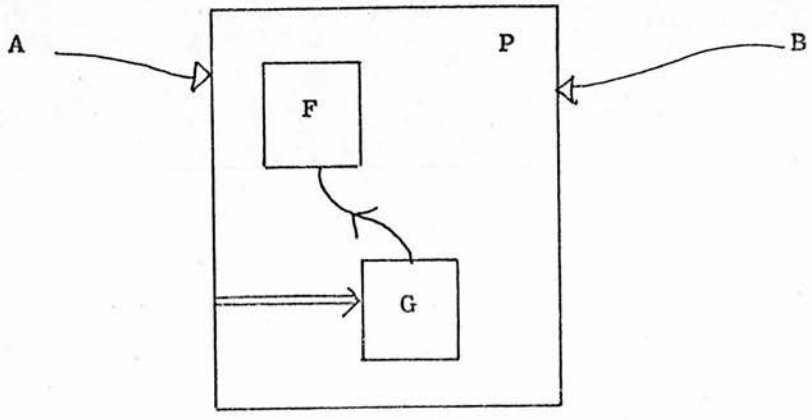



Figure 51.

In this example, P has been run at layer 2 from A, and the 1-activation G is active. G then runs another 1-activation, H, at layer 1. The situation at this stage is shown in figure 52.

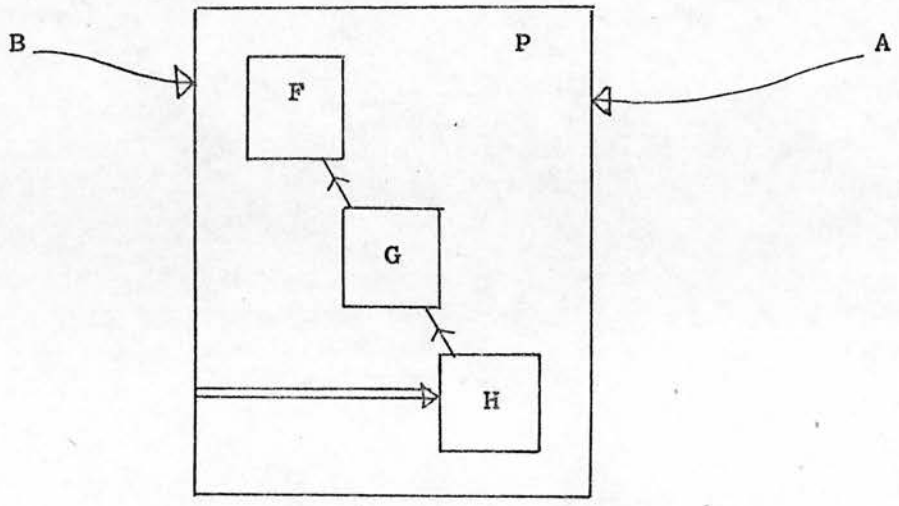



Figure 52.

Notice that the 2-activation P is side-effected. If all activity is at layer 1, the 1-activation return chain may grow and diminish within P but the effect will be noticed by all pointers to P. This is very important for it means that P represents an ongoing process.

In the particular examples we have given, the inner-outer hierarchy has been a perfect tree. If an M-activation P1 was inside P2, it was not inside any other activation of the same layer as P2. Because it is possible to run an N-layer activation at layer M (M < N), this hierarchy is not always a perfect tree. If, by convention, we removed this freedom, the hierarchy would always be tree-like. We end this section with one example resulting in a non-tree-like situation to show how complex and cumbersome the use of several layers can become. The conventions that are needed to control situations like these, and the restrictions that these conventions place on the user, show that there is something amiss with the idea of layers of control structure, in spite of the fact that many common control regimes implicitly depend on two layers.

The example is the following. A 2-activation P runs a 2-activation Q at layer 2. Q has an inner 1-activation G. The code of G thus becomes active. This is the state in figure 53.

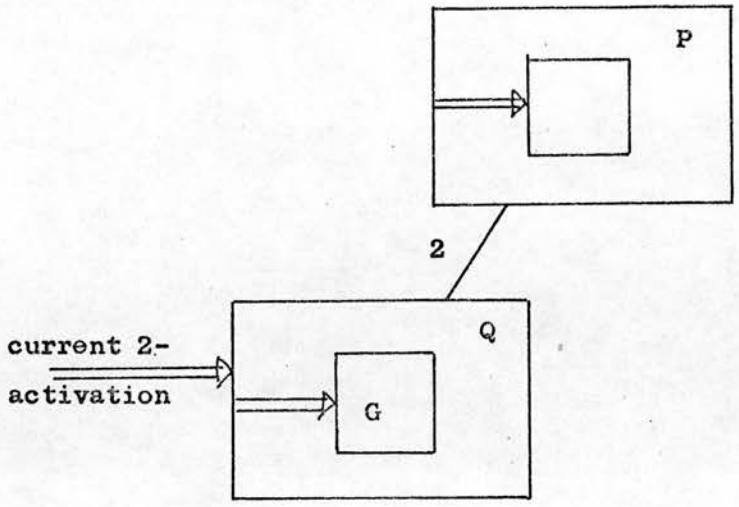



Figure 53.

G now executes a rise at layer 2 and control returns to P which stores the continuation 2-process Q in a variable A. Figure 54 shows the position now.

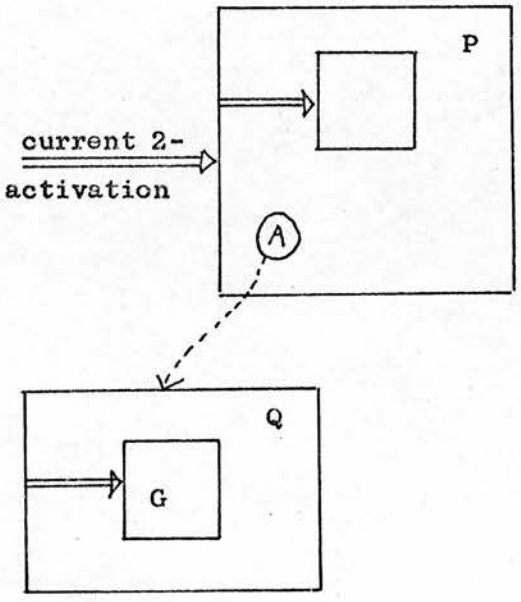



Figure 54.

P now runs at layer 2 a 2-activation R containing an inner 1-activation F, and passes Q to R as an argument. F now runs Q at layer 1. Since Q has two layers, only the inner one is used in the run. G is now an inner process of both Q and R.

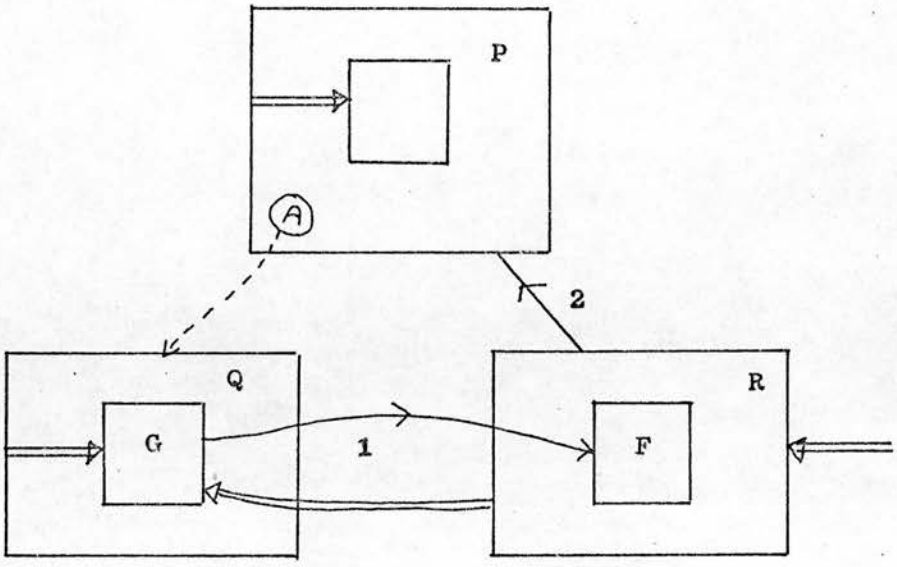



Figure 55.

9.2 A more formal definition of P-74 for arbitrary layers.

In this section we define the control and access mechanisms of P-74 more formally than in section 9.1.

A layer number is a positive integer.

An N-activation, where N is a layer number, has

1) A layer number which is N
2) An N-return pointer which references either the constant NIL or an N-activation.
3) An N-dictionary which records local variables declared at layer N in the activation.
4) If N>1 then it has an inner (N-1)-activation.
5) If N=1 then it has a SAVEDSTATE.
6) If N = 1 then it has a location, called its FST, which references the module that the activation corresponds to.

A SAVEDSTATE of a 1-activation records

1) The continuation point of the activation.
2) Temporary storage for the activation.

An N-activation is thus a nest of M-activations, one each for M=<N.

An N-process consists of all computation done while an N-activation and any of its continuations is active. An activation may have several active phases, and at the end of each it is a snapshot of an N-process.

At any time, only one N-activation, called the current N-activation, is active for each N. The current N-1 activation is thus the inner (N-1)-activation of the current N-activation. The current activations are nested at any time and together are called the current nest of N-activations or the current state.

N-activations are data structures and during execution of control primitives they will be updated and accessed. To aid in describing the operation of the control primitives we must introduce some notation for discussing our data structures.

INNER(M,<M-ACTIVATION>) ,where M < N, refers to the M-activation pointed to by the chain of inner activation pointers from the N-activation.

RET(M,<N-ACTIVATION>) ,where M < N, refers to the M-activation referenced by the M-return pointer of INNER( M, < N-ACTIVATION >).

DIC(M,<N,ACTIVATION>) ,where M < N, refers to the M-dictionary referenced by the M-dictionary pointer of INNER(M,<N-ACTIVATION>).

We also introduce a notation For updating these values. for example

X->INNER(M,<N-ACTIVATION>)

means that

INNER(M,<N-ACTIVATION>)

has been assigned to be X.

We will use this notation in two ways for each control primitive. Firstly we will describe the state of affairs after the execution of the primitive in terms of the state of affairs before. Secondly, we

will list the assignments which take place during the execution of each primitive.

The four basic control primitives are RUN, RISE, RESUME, and ENVEVAL. An execution of any control primitive is an instance of an execution of a LEAP. Each execution of a control primitive is associated with a layer number given it as argument and known as its associated layer number.

At the execution of a leap, the current nest of activations may change its structure as described separately below for each control primitive. Four N-activations are involved in any leap. They are called the tee, the target, the resultant, and the target environment. The target environment is not always involved.

The tee is the current nest of activations before the leap. It is the place control is leaving.
The resultant is the current nest of activations after the leap. It is the place control arrives at.
The target is an N-activation where N is the associated layer number of the leap.
The target environment is an N-activation where N is the associated layer number of the leap. If present, it is the environment in which the target is continued.

After any control primitive is executed the innermost N layers of the tee activation are left in the continuee.

Arguments to control primitives which are expected to be N-activations, where N is the associated layer number of this execution of the primitive, may also be N-activation specifications from which the primitive will find the required N-activation. An N-activation specification may be

1) An M-activation A, say.
If M=N then A is the required N-activation.
If M < N then A is enveloped by N-M outer layers each with NIL return pointers and empty dictionaries.
If M > N then the inner N-activation of A is the required N-activation.

2) A function module (procedure module etc) F, say.
An initial N-activation is made with F as its FST and with a SAVEDSTATE which has the first statement of F as its continuation point. All M-return pointers of the N-activation are NIL and all M-dictionaries are empty.

All four control primitives when executed with an associated layer number N obey the following relationships.

```
INNER(M,RESULTANT)==IF M>N THEN INNER(M,TEE)
                    IF M<N THEN INNER(M,TARGET)
```

No updating is done to RET(M,TEE) or DIC(M,TEE) for M/=N, nor is any updating done to RET(M,TARGET) or DIC(M,TARGET) for M<N.
These imply

```
RET(M,RESULTANT)==IF M>N THEN RET(M,TEE)
                  IF M<N THEN RET(M,TARGET)
DIC(M,RESULTANT)==IF M>N THEN DIC(M,TEE)
                  IF M<N THEN DIC(M,TARGET)
```

Furthermore the inner N layers of the tee become the continuee

```
LAYER(CONTINUEE)=N
INNER(M,CONTINUEE)==IF M=<N THEN INNER(M,TEE).
```

Expressed as a sequence of assignments these changes are

```
INNER(N,TEE)->CONTINUEE
TARGET->INNER(N,TEE)
TEE->RESULTANT.
```

In other words, there is no change to the current control structure at layers above the associated layer number. Below the associated layer number the target becomes the current activation and the tee becomes the continuee. This explains what we mean by saying that primitives at higher layers have precedence over those at lower layers. It only remains to examine the changes at exactly the associated layer. We do this next, individually for each primitive.

RUN

---

Takes <N-ACTIVATION SPEC>,<ASSOCIATED LAYER NUMBER>=>().
At a run with associated layer number N, the target is given by the first argument. The target environment is not involved.
After a run

INNER(M,RESULTANT)==IF M=N THEN TARGET.

And RET(N,TARGET) will be updated to be INNER(N,TEE).
This implies

RET(N,RESULTANT)==TARGET
DIC(N,RESULTANT)==DIC(N,TARGET).

Expressed as assignments, the operations performed at a run are

INNER(N,TEE)->CONTINUEE
TARGET->INNER(N,TEE)
TEE->RESULTANT
CONTINUEE->RET(N,RESULTANT)

RISE

---

Takes <ASSOCIATED LAYER NUMBER>=>().
At a rise with associated layer number N, the target is RET(N,TEE).
The target environment is not involved.
After a rise

INNER(M,RESULTANT)==IF M=N THEN TARGET (I.E. RET(N,TEE))

This implies

RET(N,RESULTANT)==RET(N,TARGET)==RET(N,RET(N,TEE))
DIC(N,RESULTANT)==DIC(N,TARGET)==DIC(N,RET(N,TEE)).

Expressed as assignments the operations performed at a rise are

INNER(N,TEE)->CONTINUEE
RET(N,TEE)->INNER(N,TEE)
TEE->RESULTANT

If RET(N,TEE) is NIL the target cannot be found and a rise with associated layer number one greater is executed instead.

RESUME

---

Takes <N-ACTIVATION SPEC>,<ASSOCIATED LAYER NUMBER>=>()

At a resume with associated layer number N, the target is given by the first argument. The target environment is RET(N,TEE).

After a resume

INNER(M,RESULTANT)==IF M=N THEN TARGET

RET(M,TARGET) WILL BE UPDATED TO BE THE TARGET ENVIRONMENT.

This implies

RET(N,RESULTANT)==TARGET ENVIRONMENT==RET(N,TEE)

DIC(N,RESULTANT)==DIC(N,TARGET)

Expressed as assignments the operations performed at a resume are

INNER(N,TEE)->CONTINUEE

TARGET->INNER(N,TEE)

TEE->RESULTANT

RET(N,CONTINUEE)->RET(N,RESULTANT).

If RET(N,TEE) is NIL the target environment cannot be found and a resume with associated layer number one greater is executed instead.

ENVEVAL

---

Takes <N-ACTIVATION SPEC> or <STATEMENT>, <N-ACTIVATION SPEC>, <ASSOCIATED LAYER NUMBER>=>().

The second argument is the target environment.

Case 1)

If the first argument is an N-activation-spec then it is the target.

After the enveval

INNER(M,RESULTANT)==IF M=N THEN TARGET

RET(N,TARGET) WILL BE UPDATED TO BE THE TARGET ENVIRONMENT.

This implies

RET(N,RESULTANT)==RET(N,TARGET)==TARGET ENVIRONMENT
DIC(N,RESULTANT)==DIC(N,TARGET)

Case 2)

If the first argument is a statement, then when control reaches the resultant the statement will be executed immediately and then the resultant will continue.
After the enveval

INNER(M,RESULTANT)==IF M=N THEN TARGET ENVIRONMENT

This implies

RET(N,RESULTANT)==RET(N,TARGET ENVIRONMENT)
DIC(N,RESULTANT)==DIC(N,TARGET ENVIRONMENT).

Expressed as assignments the operations performed in the two cases are
Case 1)

INNER(N,TEE)->CONTINUEE
TARGET->INNER(N,TEE)
TEE->RESULTANT
TARGET ENVIRONMENT->RET(N,RESULTANT)

Case 2)

INNER(N,TEE)->CONTINUEE
TARGET ENVIRONMENT-> INNER(N,TEE)
TEE->RESULTANT

VARIABLES
---

Variables in P-74 are denoted syntactically by P-74 identifiers. a P-74 identifier consists of two parts.

<P-74 IDENTIFIER>::=&<IDENTIFIER LAYER NUMBER><IDENTIFIER NAME>
<IDENTIFIER LAYER NUMBER>::=<POSITIVE INTEGER>
<IDENTIFIER NAME>::=<ALPHANUMERIC WORD>

Semantically, a P-74 identifier is associated with a variable in any N-activation where N is greater than the layer number of the identifier. This variable has a value, initially undefined, which can be altered by assigning to the variable. In particular, a P-74 identifier is associated with an identifier in the current state. The variable associated with a P-74 identifier, &N:<NAME> , in an N-activation A, is found by the following procedure

```
      INNER(N,A)->A;
      DIC(N,A)->D;
LOOP:DIC(N,A)->B;
      IF <NAME> HAS BEEN DECLARED IN B
      THEN RETURN WITH THE ASSOCIATED VARIABLE
      ELSE RET(N,A)->A;
          IF A=NIL
          THEN DECLARE <NAME> IN D AND RETURN WITH THE NEW
               ASSOCIATED VARIABLE
          ELSE GOTO LOOP
          CLOSE
      CLOSE
```

## 9.3 Conclusions

In this section we explain some flaws in the concept of layers. We show how automatic control regimes place restrictions on the user. We describe an alternative which avoids these restrictions. It uses a primitive which is the basis of all the control primitives of P-74 and which we call TRANSFER. Although we incorporate it at all layers, we only discuss it here at layer-1. In fact, if we removed all the other layers of P-74, but still allowed 1-activations to be manipulable items, TRANSFER by itself would be powerful enough to replace all the other primitives. TRANSFER is very similar to the message passing primitive used in ACTOR systems. We discuss this and explain the difference between the two.

Throughout this concluding section we use and develop one example beginning with the following simple problem. Suppose we require a function, SEARCH-THREE, which successively examines the elements of a list until it finds the number 3 and then returns the next element of the list as its result. In a dendrarchy, we can write SEARCH-THREE

recursively as follows.

```
FUNCTION SEARCH-THREE LIST;
IF THE LIST IS EMPTY THEN EXIT;
IF THE FIRST ELEMENT IS 3, RETURN THE SECOND ELEMENT;
OTHERWISE,DO
  SEARCH-THREE(TAIL(LIST))
END;
```

Suppose that SEARCH-THREE has returned with its first result, and that we wish to continue it so that it will find the next occurrence of 3. We will need an object to represent the current state of SEARCH-THREE. A 1-activation record is an example of such an object. The ability to manipulate a representation of the state of any process we wish to continue is the main condition that allows for general control structures. In the case of SEARCH-THREE (modified so that it can be continued), two of these objects are needed to represent the two possible return locations. One return location is the activation of SEARCH-THREE which called the current one; the other is top-level.

SIMULA is based on ALGOL which allows one return activation to correspond to any activation. This return activation is stored on a stack in ALGOL's run-time structure. The stack is protected from the user and is not accessible to him, so the activations are not manipulable. Given this situation, the natural way to extend ALGOL to provide an extra return activation is to provide CALL, DETACH, and RESUME (corresponding to P-74's RUN, RISE, and RESUME) between entire ALGOL programs. Objects representing the state of ALGOL programs (called class instances) must be made available to the user. SIMULA's control primitives, which work on these objects, have a higher precedence than the ALGOL control primitives for procedure entry and exit. If we use SIMULA's primitives, our example can be written as follows to allow SEARCH-THREE to be continued to find successive 3's (we use a mixture of english and our own syntax)

```
FUNCTION SEARCH-THREE LIST;
IF THE LIST IS EMPTY THEN EXIT;
IF THE FIRST ELEMENT IS THREE, DETACH, RETURNING THE NEXT ELEMENT;
OTHERWISE, OR ON RETURN FROM THE DETACH, DO
  SEARCH-THREE(THE-TAIL-OF-THE-LIST)
END
```

```
:INITIAL(SEARCH-THREE)->SEARCH;
:CALL SEARCH WITH [1 3 4 5 3 6] AS ARUMENT;
 THE RESULT IS 4
:CALL SEARCH AGAIN TO CONTINUE IT;
 THE RESULT IS 6:
```

We programmed P-74 in POP-2 which has a subroutine regime just as ALGOL has. However, POP-2 has a primitive which allows the user to save the current control structure of a program as an object called a saved-state. Saved-states may later be reinstated. The primitive allowed us to rectify SIMULA's fault of preventing the user from accessing 1-activations. Thus, we could overcome the restrictions of a subroutine regime consistently at both of the first two layers instead of only at the second. Using the primitive, we invented a programming trick which allowed objects representing 1-activations to be constructed. This was later used by Davies to program his POPLER 1.5 system. State-saving also allowed us to make our primitives similar at layer-1 and at layer-2. We provided RUN-1 and RUN-2 (etc). By smoothing out the disparities between these two layers, we made the concept of layers clear and this led directly to our generalisation.

Using the primitives of P-74 we can write code equivalent to the last example as follows.

```
FUNCTION SEARCH-THREE;
DECLARE LIST ARGUMENT LIST;
IF THE LIST IS EMPTY THEN <<RISE,1>>;
IF THE FIRST ELEMENT IS 3 THEN <<RISE,2,NEXT-ELEMENT>>;
OTHERWISE, OR ON RETURN FROM TOP-LEVEL, DO
  <<RUN,SEARCH-THREE,1,THE-TAIL-OF-THE-LIST>>
END;

:INITIAL(SEARCH-THREE)->SEARCH;
:<<RUN,PRINT,1,<<RUN,SEARCH,2,[1 3 4 5 3 6]>> >>;
 4
:<<RUN,PRINT,1,<<RUN,SEARCH,2>> >>;
 6:
```

We can also take advantage of the fact that 1-activations are manipulable to rewrite the program using only one layer. So

```
   FUNCTION SEARCH-THREE;
   DECLARE LIST LOCATION;ARGUMENT LIST LOCATION;
   IF THE LIST IS EMPTY THEN <<RISE,1>>;
   IF THE FIRST ELEMENT IS 3 THEN
1)   <<ENVEVAL,NO-OPERATION,LOCATION,1,
         NEXT-ELEMENT,CURRENT-1-ACTIVATION>>;
   OTHERWISE,OR ON RETURN FROM LOCATION, DO
     <<RUN,SEARCH-THREE,1,TAIL-OF-THE-LIST,LOCATION>>
   END;

   :INITIAL(SEARCH-THREE)->SEARCH;
2) :<<RUN,SEARCH,1,[1 3 4 5 3 6],CURRENT-1-ACTIVATION>>
     ->LOCATION->FIRST-RESULT-OF-SEARCH;
   :<<RUN,LOCATION,1>>
     ->LOCATION->SECOND-RESULT-OF-SEARCH;
```

Comments:

1) NO-OPERATION is executed in the activation called LOCATION. LOCATION was passed in as an argument and is a 1-activation for top-level. The current 1-activation is passed out as a result so that top-level can jump back in.
2) The search is initiated at layer-1 from top-level and the current top-level 1-activation is given as an argument together with the list to be searched.

This version only uses layer-1 primitives and layer-1 activations. The return from any activation of SEARCH-THREE to the activation which ran-1 it is done automatically by RISE-1. The return to top-level is achieved by jumping to a 1-activation which represents top-level and was passed in as an argument. When control jumps out of an activation of SEARCH-THREE to top-level, SEARCH-THREE passes back a 1-activation representing itself. Top-level uses this 1-activation to continue the search for the next occurrence of 3.

This shows that there are two ways in which control structures can be built. In one way, certain control primitives, such as RUN, manipulate return links for use by others, such as RISE. The other way is for activations to store references to other activations. This method is more powerful than the first. Using it, any convention by which control primitives automatically store references to other activations can be simulated. This includes those which correspond to all the

control regimes we discussed earlier. The only control primitive needed is one which simply transfers control and arguments from one activation to another without using or making any control links automatically. We used this primitive as the basis for programming P-74, and we call it TRANSFER. Using it, we can write our example as

```
   FUNCTION SEARCH-THREE;
   DECLARE LIST LOCATION-1 LOCATION-2;
1) ARGUMENT LIST LOCATION-1 LOCATION-2;
2) IF THE LIST IS EMPTY THEN <<TRANSFER,LOCATION-1>>;
   IF THE FIRST ELEMENT IS 3 THEN
3)   <<TRANSFER,LOCATION-2,NEXT-ELEMENT,CURRENT-1-ACTIVATION>>;
   OTHERWISE, OR ON RETURN FROM LOCATION-2, DO
4)   <<TRANSFER,SEARCH-THREE,
        TAIL-OF-THE-LIST,CURRENT-1-ACTIVATION,LOCATION-2>>;
5)   <<TRANSFER,LOCATION-1>>
   END;
```

Comments:

1) LOCATION-1 is the activation of SEARCH-THREE which called this one. LOCATION-2 is top-level.
2) If the list is empty, control returns to top-level the long way by chaining back along LOCATION-1 links.
3) The successful element is passed to top-level with the current location for continuation.
4) SEARCH-THREE is called to continue on down the list. It is given the current-1-activation and the top-level activation as its two arguments.
5) If the search of the remainder of the list ever returns, control is passed back along the chain of activations of SEARCH-THREE.

Notice that, in this example, there are no references to the relative precedences of the two return activations available to SEARCH-THREE. Instead of the system automatically controlling them under the convention that one is at a higher precedence than the other, SEARCH-THREE itself decides how it is going to use them. An activation could clearly choose from any number of activations to find which it should pass control to. It could be provided with these as arguments; it could have access to them through global variables, or it could have constructed them itself.

In more complex examples of the use of SIMULA (e. g. Dahl and Hoare, 1972, pp 196-202) the automatic control facilities are not sufficient and some of the control structure is kept as explicit references from activations to others. Particular activations from those referenced are chosen by any activation which wishes to transfer control. For example, there is no reason why SEARCH-THREE should return its results to top-level. It might instead search for 3's, 4's and 5's and return the elements following each to three different activations. Automatic control structuring facilities would be too restrictive to allow this. Languages like SIMULA and P-74 are only rich enough because they allow activations to be treated as accessible items.

We can now deal properly with the example of an inescapable loop given in section 7. 3, figure 39. The problem was that an activation record became confused when it was given two layer-1 return activations. Considering the activation to represent a process, the process is run twice and becomes obliged to return to two different places when it next executes a RISE-1. It is difficult to make a convention which always resolves this choice. On the other hand, it is possible for the particular activation which is faced with the choice to make the decision itself.

These arguments lead us to conclude that automatic control structures should not be primitive to a system. The primitive should instead be TRANSFER which can be used to simulate any automatic control structure. Syntax can be designed to make it appear that particular control structures are automatic. Such a control structure would not then be restrictive since it would always be possible to fall back on transfer when necessary.

This idea is also embodied in ACTOR systems whose primitive also passes control and a message. However, an important difference between actors and activations is that activations are side-effected when they are run whereas actors are not. Whenever an activation passes control, it represents its state at that point in its execution. If it is given control again it will continue executing its text from the point at which it left off. So an activation of F, whose code is given below, prints 1 the first time it is called, and prints 2 when it is continued. Using activations forces a distinction to be made between modules and their instances.

```
FUNCTION F;
DECLARE A;ARGUMENT A;
PRINT(1);
TRANSFER CONTROL TO A;
ARGUMENT A;
PRINT(2);
TRANSFER CONTROL TO A
END;

:INITIAL(F)->ACTIVATION;
:TRANSFER CONTROL TO ACTIVATION WITH TOP-LEVEL AS ARGUMENT;
 1
:TRANSFER CONTROL TO ACTIVATION WITH TOP-LEVEL AS ARGUMENT;
 2:
```

On the other hand, when an actor is continued, control begins again at the start of its textual definition. To achieve the effect just demonstrated we need an actor F defined as follows (the syntax is ours).

```
ACTOR F;
DECLARE A;ARGUMENT A;
PRINT(1);
TRANSFER CONTROL TO A AND PASS G THROUGH AS MESSAGE,WHERE
  G IS DEFINED AS
  ACTOR G;
  DECLARE A;ARGUMENT A;
  PRINT(2);
  TRANSFER CONTROL TO A
  END;
END;

:TRANSFER CONTROL TO F AND PASS TOP-LEVEL THROUGH AS ARGUMENT;
 1
:TRANSFER CONTROL TO THE LAST RESULT AND PASS TOP-LEVEL
    THROUGH AS ARGUMENT;
 2:
```

We conclude from our investigation of layers that some primitive like TRANSFER should be the basis of the control structure of any system which is to be used for complex control situations. The transfer primitive of P-74, and the message passing primitive of actor-

transmission are equivalent. Each can simulate the other with simple changes of programming style. With P-74, we were primarily concerned with elucidating the concept of layers. Our argument led us to unify earlier control structures and gave reasons why both P-74's transfer and ACTOR's message passing are better than these. Part I of the thesis argued for actor-like representations of knowledge so we see that the conclusions of both parts are in agreement. It is the concern of future research to apply the principles we have demonstrated to large real-world problems. This will be a demanding task.

## Appendix 1 Notes on POP-2

This appendix is intended to give a very short glimpse of the POP-2 programming language to help the reader follow some of the syntax and programming concepts assumed in the text. Further information about the language can be found in Burstall, Collins and Popplestone (1971).

POP-2 allows a user to represent and manipulate various kinds of objects including : numbers, words, strings, lists and functions. Functions are the objects which can be executed and certain primitive ones are provided in the system. The design of POP-2 is very function oriented : functions can be constructed, executed, used as arguments, passed out as results, and stored in variables.

Syntactically, there are several ways in which a function may be invoked and we mention two of them : the bracket notation and the dot notation.

Suppose we have a function of two arguments which is stored in a variable identified by F. Then we can apply it to 1 and 2 by saying

```
F(1,2);
```

The identifier preceding an open rounded bracket signifies that the value of the identifier should be applied. Arguments and results of functions are placed on a last-in-first-out stack called "the user stack". It is important to understand what happens at a function call in terms of this stack. The arguments of the function, which are within the brackets, are placed on the stack - effectively the expression inside the brackets is evaluated - then F is applied. F later takes its argument from the stack. We would have the same result if we had said

```
1,F(2);
or  1,2,F();
```

1 and 2 would have been placed on the stack and the F would be applied. In place of F we could have used any bracketed expression which evaluated to a function. So if G evaluates to a function we could apply that function to 1 by saying

```
(G())(1);
```

Instead of signifying a function application by brackets we can use a dot "." before any identifier whose value is a function. So, we could rewrite our four examples so far as

```
1,2,.F;
1,2,.F;
1,2,.F;
```
and `(.G)();`

The syntax used for assigning to a variable is the assign arrow **->**. This takes an item off the stack and assigns it to the variable following

```
e.g.  3->A;  assigns 3 to A.
       ->B;  assigns the top of the stack to B.
```

An important feature of POP-2 is that it allows list structure manipulations. We can use square brackets to construct a list of integers, words and strings at compile time - for example, a list of four elements is

```
[THE CAT 3 'SAT']
```

Alternatively, we can construct a list by evaluating an expression at execute time. The notation for this uses decorated square brackets.

```
e.g.  [% 3,CAT, F(1,2) %]
```

In this example the second element will be the value of CAT whereas in the first example it was the word 'CAT'.

There are two standard functions HD and TL which take a list as argument and produce as results, respectively, the first element of the list and the remainder of the list.

```
Thus    HD([1 2 3]);  produces 1
and     TL([1 2 3]);  produces [2 3].
```

Now we can introduce the idea of doublets. All functions can be given an associated function called an updater. A function which has an updater is called a doublet. The function itself is applied in the normal way but the updater is applied when it is the top-level function immediately to the right of "->". Thus

```
3->HD([1 2 3]);
```

would put 3 on the stack, put [1 2 3] on the stack and apply the updater of HD. This updater takes two arguments off the stack and would alter the first element of [1 2 3] to be 3. This is another use of the assign arrow.

New functions can be defined using LAMBDA or FUNCTION. The syntax of a LAMBDA expression is approximately

```
LAMBDA <formal parameter list>; <function body> end;
```

This leaves a new function on the stack. The function will take arguments off the stack and name them as specified in the formal parameter list. It will then execute the function body. FUNCTION is used to produce functions and name them. So

```
FUNCTION <function name><formal parameter list>; <function body> end;
```

will declare a variable <function name> and assign the new function to that variable.

There is another feature called partial application which we must describe in slightly greater detail since we use it heavily in the body of the thesis.

Functions normally take their arguments off the stack while they are executed. In POP-2 it is possible to permanently associate arguments with a function to make a new function called a closure. When a closure function is applied it first unloads its stored arguments onto the stack and then applies its function part. We call the associated arguments of a closure the frozen arguments and refer to the operation of associating arguments as either partially applying a function to some arguments, or else freezing some

arguments into a function. Decorated round brackets are used to signify partial application. Thus

```
PR(% 3 %);
```

leaves a function which will print 3, on the stack, and

```
LAMBDA F; F(); F() END(%PR(%3%)%);
```

leaves a function which will print 3 twice.

Partial application is very important in POP-2. In ALGOL, variables are textually bound so it is possible to see from a listing of a program what variable any identifier refers to. In POP-2 and LISP, however, variables are dynamically bound. An identifier refers to a particular variable because of the run-time context. This is often an advantage, but in the cases where we want to circumvent this we use partial application. The problem arises when we wish to write functions which produce functions as results or pass them as arguments. It is a famous problem referred to as the Funarg problem (Moses, 1970). When a function is defined in LISP, we specify whether all its identifiers refer to variables in their execute time environment. We have the choice only of entire environments, either define time or execute time.

In POP-2, the second of these is the usual way though we can selectively freeze values into any new function. Suppose, for example, we wished to write a function F which produced a function which referred to the current value of the variable A. We do this by partial application as follows

```
FUNCTION F;
LAMBDA A;  <body>   END(%A%)
END;
```

This freezes the current value of A into the function. Notice that it freezes the value and not the variable. To achieve the effect of freezing in the variable, we would need to make A into a reference.

Bibliography

ANDERSON, B. (1972) "Programming Languages for Artificial Intelligence : The Role of Nondeterminism". Experimental Programming Reports : No. 25. School of Artificial Intelligence, University of Edinburgh.

ANDERSON, D.B. and HAYES, P.J. (1971) "The Logician's Folly". Department of Computational Logic, University of Edinburgh. Memo No. 54.

AUSTIN, J.L. (1962) "How to Do Things with Words". Oxford University Press.

BORROW, D.G. and WEGBREIT, B. (1972) "A Model and Stack Implementation of Multiple Environments". BBN Report, Bolt Beranek and Newman Inc. Cambridge Massachusetts. Also (1973) C.A.C.M. Vol. 16/8 p. 519.

BOYER, R.S. and MOORE, J.S. (1972) "Proving Theorems about LISP Functions". D.C.L. Memo 60. School of Artifical Intelligence, University of Edinburgh.

BROWN, J.S., BURTON, R.R. and ZDYBEL, F. (1973) "A Model-Driven Question-Answering System for Mixed Initiative Computer-Assisted Instruction". IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-3, No. 3.

BURSTALL, R.M., COLLINS, J.S., and POPPLESTONE, R.J. (1971) "Programming in POP-2". Edinburgh University Press.

CARBONELL, J.R. (1970) "Mixed Initiative Man-Computer Instructional Dialogues". Ph.D. Dissertation, M.I.T., Cambridge, Mass.

CARBONELL, J.R. (1970) "AI in CAI : An Artificial Intelligence Approach to Computer Assisted Instruction". IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4.

CHARNIAK, E. (1972) "Towards a Model of Children's Story Comprehension". Ph.D. Thesis. AI-TR-266. Massachusetts Institue of Technology.

COLES, L.S. (1972) "Techniques for information retrieval using an inferential question answering system with natural language input." S.R.I. Technical memo 74, (November), Stanford, California.

CONWAY, J.H. (1970) "The Game of Life" in "Mathematical Games". Martin Gardner, Scientific American, October 1970.

CONWAY, M.E. (1963) "Design of a Separable Transition - Diagram Compiler". C.A.C.M. Vol. 6/7/1963 p.396.

DAHL, O-J., and HOARE, C.A.R. (1972) "Hierarchical Program Structures". In "Structured Programming" (Dahl, O-J., Dijkstra, E.W., and HOARE, C.A.R.) A.P.I.C. Studies in Data Processing, No. 8 Academic Press.

DAHL, O-J., and NYGAARD, K. (1966) "SIMULA - An ALGOL based Simulation Language". C.A.C.M. 9, 671 - 678.

DAVIES, D.J.M. (1971) "POPLER : A POP-2 PLANNER". MIP-R-89. School of Artificial Intelligence, University of Edinburgh.

DAVIES, D.J.M. (1973) "POPLER 1.5 Reference Manual". T.P.U. Report No. 1. School of Artificial Intelligence, University of Edinburgh.

DAVIES, D.J.M. (1974) "Representing Negation in a PLANNER system", A.I.S.B. Summer Conference, University of Sussex.

DIJKSTRA, E.W. (1960) "Recursive Programming". Numerische Mathematik 2, 312, 318. Also in "Programming Systems and Languages", S. Rosen (Ed.), McGraw-Hill, New York, 1967.

FARBER, D.J., GRISWOLD, R.W., and POLONSKY, I.P. (1964) "SNOBOL, a string manipulation language". J.A.C.M., 1964, Vol. 11, pp. 21-30.

FLOYD, R.W. (1967) "Non deterministic Algorithms". J.A.C.M. 14 (October), pp. 636-644.

FILLMORE, C.J. (1968) "The Case for Case" in "Universals in Linguistic Theory". Bach and Harms (Eds.), New York : Holt Rinehart and Winston.

GOLOMB, S.W. and BAUMERT, L.D. (1965) "Backtrack Programming", J.A.C.M. 12 (October), pp. 516-524.

HALLIDAY, M.A.K. (1970) "Language Structure and Language Function". In New Horizons in Linguistics", Ed. Lyons, I. Pelican.

HEWITT, C. (1969) "PLANNER" : A Language for Manipulating Models and Proving Theorems in a Robot". Proceedings of the International Joint Conference on Artificial Intelligence Washington D.C.

HEWITT, C. (1972) "Descriptions and Theoretical Analysis (using Schemata) of PLANNER : A Language for Proving Theorems and Manipulating Models in a Robot". Ph.D. Thesis. AI-TR-258.

HEWITT, C. (1973) "ACTOR Induction and Meta-evaluation". Proc. A.C.M. SIGACT-SIGPLAN Conference, Boston (October 1973).

HEWITT, C., BISHOP, P., and STEIGER, R. (1973) "A Universal Modular ACTOR Formalism for Artificial Intelligence". Proc. 3rd IJCAI (1973), pp. 235-245.

HOARE, C.A.R. (1972) "Notes on Data Structuring". In "Structured Programming" (Dahl, O-J., Dijkstra, E.W., and Hoare, C.A.R)., A.P.I.C. Studies in Data Processing, No. 8, Academic Press.

HOARE, C.A.R. (1972) "Record Handling". In Programming Languages, (Ed. Genuys, F.) Academic Press, London, pp. 291-341.

HOWE, J.A.M. (1973) "Individualizing Computer Aided Instruction". In "Artificial and Human Thinking", eds. Elithorn and Jones, Elsevier Press.

KISS, G.R. (1974) "Information Processing Structures". Speech and Communication Research Unit, University of Edinburgh.

KNAPMAN, J. (1973) "PROCESS 1.5 Description and User's Guide". Bionics Research Reports : No. 11 (1973). School of Artificial Intelligence, University of Edinburgh.

KNUTH, D.E. (1968) "The Art of Computer Programming", Vol. 1, "Fundamental Algorithms". Addison-Wesky.

LANDIN, P.J. (1965) "A Generalisation of Jumps and Labels", Univac Systems Programming Research.

LONGUET-HIGGINS, H.C. (1972) "The Algorithmic Description of Natural Language". Review Lecture, Proc R. Soc. Lond. B.182, 255-276.

McCARTHY, J. (1970) "A Basis for a Mathematical Theory of Computation" In "Computer Programming and Formal Systems". Eds. Braffort, P. and Hirschberg, D., North-Holland.

McCARTHY, J., ABRAHAMS, P.W., EDWARDS, D.J., HART, T.P., and LEVIN, M.I. (1962) "LISP 1.5 Programmer's Manual", M.I.T. Press.

McDERMOTT, D.V. and SUSSMAN, G.J. (1972) "The Conniver Reference Manual". AI Memo 259. AI LAB., Mass. Inst. Technology, Cambridge, Massachusetts.

MINSKY, M. (1967) "Computation : Finite and Infinite Machines". Prentice-Hall.

MINSKY, M. (1974) "A Framework for Representing Knowledge", M.I.T. AI Memo No. 306, June.

MINSKY, M, and PAPERT, S. (1972) "Artificial Intelligence. Progress Report". AI Memo 252. Massachusetts Institute of Technology, Cambridge, Massachusetts.

MOORE, J S. (1973) "Computational Logic : Structure Sharing and Proof

of Program Properties", Part II. D.C.L. Memo No. 68, Department of Artificial Intelligence, Edinburgh (Ph.D. Thesis).

MOSES, J. (1970) "The Function of FUNCTION in LISP". SIGSAM Bulletin, (July, 1970), pp. 13-27.

O'SHEA, T. (1973) "Some Experiments with an Adaptive Self-Improving Teaching System". Department of Computer Science and CAI Lab., University of Texas. Technical Report NL18.

PAPERT, S. (1973) "Uses of Technology to Enhance Education". Proposal to the National Science Foundation. M.I.T., Cambridge, Mass.

POWER, R. (1974) "A Computer Model of Conversation". Ph.D. Thesis, University of Edinburgh.

QUILLIAN, M.R. (1968) "Semantic Memory", in "Semantic Information Processing". Ed. M. Minsky. M.I.T. Press, Cambridge, Mass.

RAPHAEL, B. (1968) "A Computer Program for Semantic Information Retrieval". In "Semantic Information Processing". Ed. M.Minsky, M.I.T. Press, Cambridge, Mass.

REYNOLDS. J. (1970) "GEDANKEN - A simple Typeless Language Based on the principle of Completeness and the Reference Concept". CACM, Vol. 13, No. 5, (May, 1970), pp. 308-319.

SAMUEL, A. (1959) "Some Studies in Machine Learning using the Game of Checkers". IBM J. Res. Develop. 3 (3) (July, 1959), pp. 211-229.

SELF, J.A. (1973) "Student Models in Computer-Aided Instruction". Centre for Computer Studies, University of Leeds.

SIMMONS, R.F. (1973) "Semantic Networks : Their Computation and Use for Understanding English Sentences". In "Computer Models of Thought and Language". Schank and Colby (Eds.). Freeman.

SLOMAN, A. (1972) "Some Steps Toward the Design of a Mind : A Bird's Eye View". Department of Computational Logic, University of Edinburgh. Memo No. 59.

SMITH, B., and HEWITT, C. (1973) "Towards a Programming Apprentice", A.I.S.B. proceedings, Brighton.

STANSFIELD, J.L. (1971) "A Dialogue Teaching Program". CAI-6 Bionics Research Laboratory, University of Edinburgh.

STANSFIELD, J.L. (1972) Unpublished Progress Paper. Bionics Research Laboratory, University of Edinburgh.

STANSFIELD, J.L. (1972) [PROCESS 1] : A Generalisation of Recursive Programming Languages". Bionics Research Report No. 8, School of Artificial Intelligence, University of Edinburgh.

STANSFIELD, J.L. (1974) "Artificial Intelligence and Real Learning". Bionics Memo CAI-16. School of Artificial Intelligence, University of Edinburgh.

SUSSMAN, G.J. and McDERMOTT, D. (1972) "Why Conniving is better than PLANNING". Proc. 1972 FJCC, Vol. 41, Part II, pp. 1171-1179. Also, AI Memo No. 255A. M.I.T., Cambridge, Mass.

SUSSMAN, G.J. (1973) "A Computational Model of Skill Acquisition". Ph.D. Thesis, M.I.T., Cambridge, Mass.

WEGNER, P. (1971) "Programming Languages, Information Structures, and Machine Organisation". McGraw-Hill.

WINOGRAD, T. (1971) "Understanding Natural Language". M.I.T. Ph.D. Thesis. Also (1972) Edinburgh University Press. Also (1972) New York : Academic Press.

WINOGRAD, T. (1974) "Frame Representations and the Declarative/Procedural Controversy". Draft version. Stanford AI Lab., April.

WINSTON, P.M. (1970) "Learning Structural Descriptions from Examples". Ph.D. Thesis. AI-TR-231. M.I.T., Cambridge.

WITTGENSTEIN, L. (1953) "Philosophical Investigations". Third Edition (1958), pub. Blackwell, Oxford.